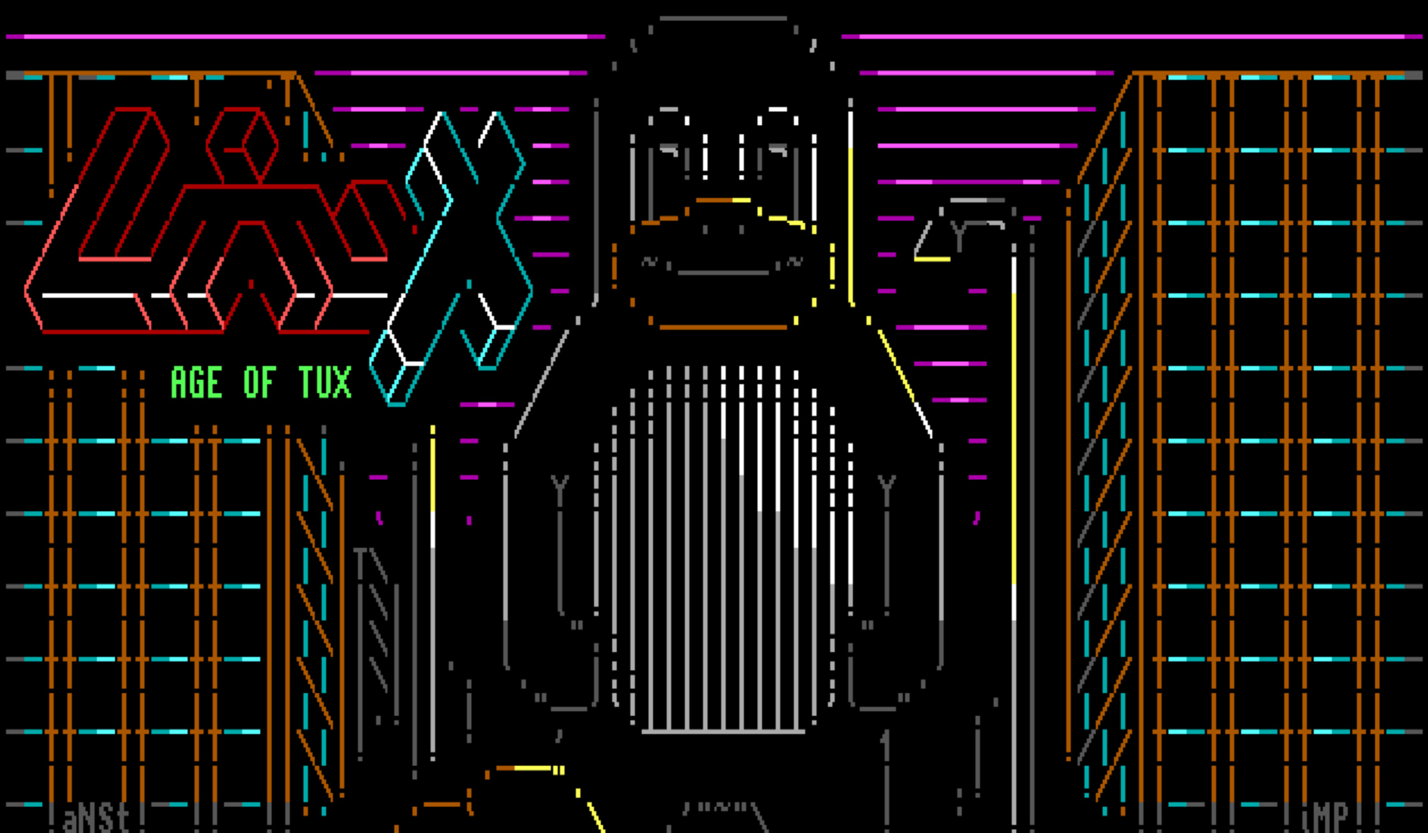
Using OpenAI's GPT-2

Mastodon and IRC

Introducing the Yocto Project

LINUX JOURNAL

Since 1994: The original magazine of the Linux community



The Command Line

>> Regular Expressions Primer

>> Scripting with Tcl

>> Command-Line Video Game Roundup

>> How to Live Entirely in the Terminal

ISSUE 300 | JULY 2019
www.linuxjournal.com

CONTENTS | JULY 2019 ISSUE 300

66 DEEP DIVE: THE COMMAND LINE



67 A Guide to Basic Command-Line Tasks

by Dave Taylor

A whirlwind tour of the "user interface that wouldn't die".

80 Without a GUI—How to Live Entirely in a Terminal

by Bryan Lunduke

Sure, it may be hard, but it is possible to give up graphical interfaces entirely—even in 2019.

90 How to Expand Your Command-Line Scripting Options with Tcl

by Mitch Frazier

Get started scripting with Tcl, the Tool Command Language—this actually is your father's Oldsmobile.

118 Regular Expressions: the Linux User's Second Language

by Andrew Piziali

What are "regular expressions", and why should you bother learning them? This article answers those questions and more.

131 The Best Command-Line-Only Video Games

by Bryan Lunduke

A rundown of the biggest, most expansive and impressive games that you can run entirely in your Linux shell.

CONTENTS

6 **The Command-Line Issue**
by Bryan Lunduke

10 **From the Editor**
by Doc Searls
In the End Is the Command Line

16 **Letters**

UPFRONT

21 **GIS on Linux with SAGA**
by Joey Bernard

27 **Patreon and *Linux Journal***

28 **Lessons in Vendor Lock-in: Google and Huawei**
by Kyle Rankin

32 **Reality 2.0: a *Linux Journal* Podcast**

33 **ASCII Art Contest**

35 **News Briefs**

COLUMNS

39 **Kyle Rankin's Hack and /**
What Really IRCs Me: Mastodon

45 **Reuven M. Lerner's At the Forge**
Python's Mypy: Callables and Generators

53 **Dave Taylor's Work the Shell**
Bash Shell Games: Let's Play *Go Fish!*

60 **Zack Brown's diff -u**
What's New in Kernel Development

158 **Glyn Moody's Open Sauce**
Online Censorship Is Coming—Here's How to Stop It

LINUX JOURNAL (ISSN 1075-3583) is published monthly by Linux Journal, LLC. Subscription-related correspondence may be sent to 9597 Jones Rd, #331, Houston, TX 77065 USA. Subscription rate is $34.50/year. Subscriptions start with the next issue.

ARTICLES

140 An AI Wizard of Words
by Marcel Gagné
A look at using OpenAI's Generative Pretrained Transformer 2 (GPT0-2) to generate text.

150 Linux IoT Development: Adjusting from a Binary OS to the Yocto Project Workflow
by Mirza Krak
Introducing the Yocto Project and the benefits of using it in embedded Linux development.

AT YOUR SERVICE

SUBSCRIPTIONS: *Linux Journal* is available as a digital magazine, in PDF, EPUB and MOBI formats. Renewing your subscription, changing your email address for issue delivery, paying your invoice, viewing your account details or other subscription inquiries can be done instantly online: https://www.linuxjournal.com/subs. Email us at subs@linuxjournal.com or reach us via postal mail at *Linux Journal*, 9597 Jones Rd #331, Houston, TX 77065 USA. Please remember to include your complete name and address when contacting us.

ACCESSING THE DIGITAL ARCHIVE: Your monthly download notifications will have links to the different formats and to the digital archive. To access the digital archive at any time, log in at https://www.linuxjournal.com/digital.

LETTERS TO THE EDITOR: We welcome your letters and encourage you to submit them at https://www.linuxjournal.com/contact or mail them to *Linux Journal*, 9597 Jones Rd #331, Houston, TX 77065 USA. Letters may be edited for space and clarity.

SPONSORSHIP: We take digital privacy and digital responsibility seriously. We've wiped off all old advertising from *Linux Journal* and are starting with a clean slate. Ads we feature will no longer be of the spying kind you find on most sites, generally called "adtech". The one form of advertising we have brought back is sponsorship. That's where advertisers support *Linux Journal* because they like what we do and want to reach our readers in general. At their best, ads in a publication and on a site like *Linux Journal* provide useful information as well as financial support. There is symbiosis there. For further information, email: sponsorship@linuxjournal.com or call +1-360-890-6285.

WRITING FOR US: We always are looking for contributed articles, tutorials and real-world stories for the magazine. An author's guide, a list of topics and due dates can be found online: https://www.linuxjournal.com/author.

NEWSLETTERS: Receive late-breaking news, technical tips and tricks, an inside look at upcoming issues and links to in-depth stories featured on https://www.linuxjournal.com. Subscribe for free today: https://www.linuxjournal.com/enewsletters.

LINUX JOURNAL

EDITOR IN CHIEF: Doc Searls, doc@linuxjournal.com

EXECUTIVE EDITOR: Jill Franklin, jill@linuxjournal.com

DEPUTY EDITOR: Bryan Lunduke, bryan@lunduke.com

TECH EDITOR: Kyle Rankin, lj@greenfly.net

ASSOCIATE EDITOR: Shawn Powers, shawn@linuxjournal.com

EDITOR AT LARGE: Petros Koutoupis, petros@linux.com

CONTRIBUTING EDITOR: Zack Brown, zacharyb@gmail.com

SENIOR COLUMNIST: Reuven Lerner, reuven@lerner.co.il

SENIOR COLUMNIST: Dave Taylor, taylor@linuxjournal.com

PUBLISHER: Carlie Fairchild, publisher@linuxjournal.com

ASSOCIATE PUBLISHER: Mark Irgang, mark@linuxjournal.com

DIRECTOR OF DIGITAL EXPERIENCE:
Katherine Druckman, webmistress@linuxjournal.com

DIRECTOR OF SALES: Danna Vedder, danna@linuxjournal.com

GRAPHIC DESIGNER: Garrick Antikajian, garrick@linuxjournal.com

COVER IMAGE: aNACHRONiST (Daniel Kelly)

ACCOUNTANT: Candy Beauchamp, acct@linuxjournal.com

Linux Journal is published by, and is a registered trade name of,
Linux Journal, LLC. 4643 S. Ulster St. Ste 1120 Denver, CO 80237

SUBSCRIPTIONS
E-MAIL: subs@inuxjournal.com
URL: www.linuxjournal.com/subscribe
Mail: 9597 Jones Rd, #331, Houston, TX 77065

SPONSORSHIPS
E-MAIL: sponsorship@linuxjournal.com
Contact: Director of Sales Danna Vedder
Phone: +1-360-890-6285

THE COMMAND-LINE ISSUE

By Bryan Lunduke

Summer. 1980-something. An elementary-school-attending, *Knight Rider*-T-Shirt-wearing version of myself slowly rolls out of bed and shuffles to the living room. There, nestled between an imposingly large potted plant and an over-stocked knick-knack shelf, rested a beautifully gray, metallic case powered by an Intel 80286 processor—with a glorious, 16-color EGA monitor resting atop.

This was to be my primary resting place for the remainder of the day: in front of the family computer.

That PC had no graphical user interface to speak of—no X Window System, no Microsoft Windows, no Macintosh Finder. There was just a simple command line—in this case, MS-DOS. (This was long before Linux became a thing.) Every task I wished to perform—executing a game, moving files—required me to type the commands in via a satisfyingly loud, clicky keyboard. No, "required" isn't the right word here. Using the computer was a joy. "Allowed" is the right word. I was *allowed* to enjoy typing those commands in. I never once resented that my computer needed to be interacted with via a keyboard. That is, after all, what



Bryan Lunduke is a former Software Tester, former Programmer, former VP of Technology, former Linux Marketing Guy (tm), former openSUSE Board Member... and current Deputy Editor of *Linux Journal*, Marketing Director for Purism, as well as host of the popular *Lunduke Show*. More details: http://lunduke.com.

computers do. That's what they're for—you type in commands, and the computer executes them for you, often with a "beep".

For a kid, this was empowering—taking my rudimentary understanding of language (aided, at first, by a handy DOS command cheat sheet) and weaving together strings of words that commanded the computer to do my bidding. It was like organizing runes to enact an ancient spell. It was magic. And I was a wizard. Did I miss not being able to "double click" or "drag and drop"? Of course not. I'd seen some such, mouse-driven user interfaces (like the early Macintoshes), but—from my vantage—that wasn't how computers really worked. I viewed such things as cool-looking, but not necessary. Computers use words. Powerful, magical words.

But this isn't 1980-something. In fact, it's barely 2010-something. (Did anyone else just realize that it's almost 2020?) For better or worse, how people use—and view—computers has changed dramatically since the days of *Knight Rider*. Modern operating systems are, often, belittled if they require users to interact with the machine via a command line. The graphical user interface is king. Which is, perhaps, the inevitable evolution of how we all interact with our computers.

Yet the value of the command line (or terminal, shell and so on) is still there. For many, it makes using computers more accessible. For others, it provides streamlined workflows that a mouse or touch-driven interface simply can't compete with. And, for others still, the blinking cursor provides a bit of nostalgic joy—or an aesthetically simple, and distraction-free, environment.

This issue of *Linux Journal* celebrates the cursor—that wonderful blinking underscore and all the potential that it holds.

To get warmed up, Dave Taylor (author of *Learning Unix for Mac OS X* and *Wicked Cool Shell Scripts*) starts off with a whirlwind tour of the "the user interface that wouldn't die" in his article "A Guide to Basic Command-Line Tasks".

Then, your favorite *Linux Journal* Deputy Editor takes you on a tour of command-line applications that you can use as replacements for some of the most common graphical tools in "Without a GUI—How to Live Entirely in a Terminal in 2019".

Once you've gotten a feel for basic shell commands and experience doing some common computing tasks entirely from the terminal, it's time to up your game. In "How to Expand Your Command-Line Scripting Options with Tcl", Mitch Frazier gives an introduction to some more advanced scripting with the Tool Command Language.

What about regular expressions? Every Linux aficionado needs, eventually, to get a basic regex primer. Maybe you'll love it; maybe you'll hate it. But, either way, it's a rite of passage for Linux users since the dawn of time. Andrew Piziali provides exactly that in his excellent "Regular Expressions: the Linux User's Second Language".

Phew.

Finally, to reward yourself for expanding your command-line knowledge and powers, let's play some video games. We've pulled together the biggest, most expansive and impressive command-line-only video games in the aptly titled "The Best Command-Line-Only Video Games". No 3D, no VR, no full motion video—just good old-fashioned ASCII characters flying around your screen.

When all is said and done, the Linux-powered computer in front of me is so much more powerful than that 1980-something computer—it borders on the ridiculous. But there are some things to be learned and admired about the text-only computing heritage—both about how computing used to be and how you can better enjoy and utilize the computers as they are today. ■

Send comments or feedback
via https://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

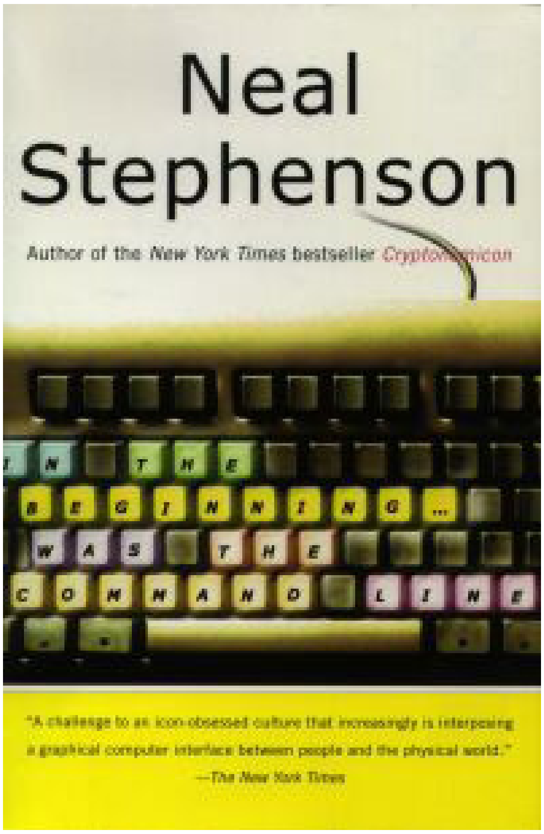
In the End Is the Command Line

Times have changed every character but one in Neal Stephenson's classic. That one is Linux.

By Doc Searls



Doc Searls is a veteran journalist, author and part-time academic who spent more than two decades elsewhere on the *Linux Journal* masthead before becoming Editor in Chief when the magazine was reborn in January 2018. His two books are *The Cluetrain Manifesto*, which he co-wrote for Basic Books in 2000 and updated in 2010, and *The Intention Economy: When Customers Take Charge*, which he wrote for Harvard Business Review Press in 2012. On the academic front, Doc runs ProjectVRM, hosted at Harvard's Berkman Klein Center for Internet and Society, where he served as a fellow from 2006–2010. He was also a visiting scholar at NYU's graduate school of journalism from 2012–2014, and he has been a fellow at UC Santa Barbara's Center for Information Technology and Society since 2006, studying the internet as a form of infrastructure.



I was wandering through Kepler's, the legendary bookstore, sometime late in 1999, when I spotted a thin volume with a hard-to-read title on the new book table. *In the Beginning...Was the Command Line*, the cover said.

The command line was new to me when I started writing for *Linux Journal* in 1996. I hadn't come from UNIX or from programming. My tech background was in ham radio and broadcast engineering, and nearly all my hacking was on RF hardware. It wasn't a joke when I said the only code I knew was Morse. But I was amazed at how useful and necessary the command line was, and I was

thrilled to see Neal Stephenson was the author of that book. (Pro tip: you can tell the commercial worth of an author by the size of his or her name on the cover. If it's bigger than the title of the book, the writer's a big deal. Literally.)

So I bought it, and then I read it in one sitting. You can do the same. In fact, I command that you do, if you haven't already, because (IMHO) it's the most classic book ever written about both the command line and Linux itself—a two-fer of the first order.

And I say this in full knowledge (having re-read the whole thing many times, which is easy, because it's short) that much of what it brings up and dwells on is stale in the extreme. The MacOS and the Be operating systems are long gone (and the Be computer was kind of dead on arrival), along with the Windows of that time. Today Apple's OS X is BSD at its core, while Microsoft produces lots of open-source code and contributes mightily to The Linux Foundation. Some of Neal's observations and complaints about computing and the culture of the time also have faded in relevance, although some remain enduringly right-on. (If you want to read a section-by-section critique of the thing, Garrett Birkel produced one in the mid-2000s with Neal's permission. But do read the book first.)

What's great about *Command Line* is how well it explains the original virtues of UNIX, and of Linux as the operating system making the most of it:

> The file systems of Unix machines all have the same general structure. On your flimsy operating systems, you can create directories (folders) and give them names like Frodo or My Stuff and put them pretty much anywhere you like. But under Unix the highest level—the root—of the filesystem is always designated with the single character "/" and it always contains the same set of top-level directories:
>
> - /usr
> - /etc
> - /var
> - /bin
> - /proc

- /boot
- /home
- /root
- /sbin
- /dev
- /lib
- /tmp

and each of these directories typically has its own distinct structure of subdirectories. Note the obsessive use of abbreviations and avoidance of capital letters; this is a system invented by people to whom repetitive stress disorder is what black lung is to miners. Long names get worn down to three-letter nubbins, like stones smoothed by a river.

This is not the place to try to explain why each of the above directories exists, and what is contained in it. At first it all seems obscure; worse, it seems deliberately obscure. When I started using Linux I was accustomed to being able to create directories wherever I wanted and to give them whatever names struck my fancy. Under Unix you are free to do that, of course (you are free to do anything) but as you gain experience with the system you come to understand that the directories listed above were created for the best of reasons and that your life will be much easier if you follow along (within /home, by the way, you have pretty much unlimited freedom).

After this kind of thing has happened several hundred or thousand times, the hacker understands why Unix is the way it is, and agrees that it wouldn't be the same any other way. It is this sort of acculturation that gives Unix hackers their confidence in the system, and the attitude of calm, unshakable, annoying superiority captured in the Dilbert cartoon. Windows 95 and MacOS are products, contrived by engineers in the service of specific companies. Unix, by contrast, is not so much a product as it is a painstakingly compiled oral history of the hacker subculture. It is our Gilgamesh epic.

What made old epics like Gilgamesh so powerful and so long-lived was that they were living bodies of narrative that many people knew by heart, and told over

> and over again—making their own personal embellishments whenever it struck their fancy. The bad embellishments were shouted down, the good ones picked up by others, polished, improved, and, over time, incorporated into the story. Likewise, Unix is known, loved, and understood by so many hackers that it can be re-created from scratch whenever someone needs it. This is very difficult to understand for people who are accustomed to thinking of OSes as things that absolutely have to be bought.
>
> Many hackers have launched more or less successful re-implementations of the Unix ideal. Each one brings in new embellishments. Some of them die out quickly, some are merged with similar, parallel innovations created by different hackers attacking the same problem, others still are embraced, and adopted into the epic. Thus Unix has slowly accreted around a simple kernel and acquired a kind of complexity and asymmetry about it that is organic, like the roots of a tree, or the branchings of a coronary artery. Understanding it is more like anatomy than physics.

There are many other yummy passages. Here's one example:

> Documentation, under Linux, comes in the form of man (short for manual) pages. You can access these either through a GUI (xman) or from the command line (man). Here is a sample from the man page for a program called rsh:
>
> "Stop signals stop the local rsh process only; this is arguably wrong, but currently hard to fix for reasons too complicated to explain here."
>
> The man pages contain a lot of such material, which reads like the terse mutterings of pilots wrestling with the controls of damaged airplanes. The general feel is of a thousand monumental but obscure struggles seen in the stop-action light of a strobe. Each programmer is dealing with his own obstacles and bugs; he is too busy fixing them, and improving the software, to explain things at great length or to maintain elaborate pretensions.
>
> In practice you hardly ever encounter a serious bug while running Linux. When you do, it is almost always with commercial software (several vendors sell

> software that runs under Linux). The operating system and its fundamental utility programs are too important to contain serious bugs. I have been running Linux every day since late 1995 and have seen many application programs go down in flames, but I have never seen the operating system crash. Never. Not once. There are quite a few Linux systems that have been running continuously and working hard for months or years without needing to be rebooted.
>
> Commercial OSes have to adopt the same official stance towards errors as Communist countries had towards poverty. For doctrinal reasons it was not possible to admit that poverty was a serious problem in Communist countries, because the whole point of Communism was to eradicate poverty. Likewise, commercial OS companies like Apple and Microsoft can't go around admitting that their software has bugs and that it crashes all the time, any more than Disney can issue press releases stating that Mickey Mouse is an actor in a suit.
>
> This is a problem, because errors do exist and bugs do happen. Every few months Bill Gates tries to demo a new Microsoft product in front of a large audience only to have it blow up in his face. Commercial OS vendors, as a direct consequence of being commercial, are forced to adopt the grossly disingenuous position that bugs are rare aberrations, usually someone else's fault, and therefore not really worth talking about in any detail. This posture, which everyone knows to be absurd, is not limited to press releases and ad campaigns. It informs the whole way these companies do business and relate to their customers. If the documentation were properly written, it would mention bugs, errors, and crashes on every single page. If the on-line help systems that come with these OSes reflected the experiences and concerns of their users, they would largely be devoted to instructions on how to cope with crashes and errors.
>
> But this does not happen. Joint stock corporations are wonderful inventions that have given us many excellent goods and services. They are good at many things. Admitting failure is not one of them. Hell, they can't even admit minor shortcomings.

I could go on, but I'd rather have you read the book than give more away.

While various sources online (**Wikipedia included**) say Neal probably won't update the book, **his own page on the book** says:

> This was originally just a set of musings about Graphical User Interfaces (GUIs) that gradually took on the shape of an essay. On the spur of the moment, the decision was made to post it on my publisher's website. Today we would say that it went viral, but back then we said that it had been Slashdotted. Anyway, the resulting traffic broke the publisher's servers until various readers set up mirror sites to handle the load. This essay is in need of an update, which I'm slowly working on, but there is still some material in here that many readers might find interesting.

I believe at least some of those many are *Linux Journal* readers.

If you want to buy the book, **here's the Amazon link**. If you want to read the whole thing online, **look it up**. (I count four complete copies in the first page of search results.)

And Neal, let us know when that update is ready. It'll be news, and we want the scoop. ■

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

One for Doc

Doc Searls should find this story of interest: "Data protection watchdog launches statutory inquiry into Google's Ad Exchange".

—Paul Barry

Doc Searls replies: Thanks, Paul. Indeed, there are many snares around the ankles of Google and Facebook. I also notice that RTE itself is also tracking me, or trying to. According to Privacy Badger, all of these are "suspected trackers" as well, all meant to be escorted into my browser by that story:

- logws1309.ati-host.net
- cdnjs.cloudflare.com
- securepubads.g.doubleclick.net
- connect.facebook.net
- www.google-analytics.com
- adservice.google.com
- www.googletagservices.com
- sb.scorecardresearch.com

Their participation in the same icky system as Google, Facebook and other easy targets is a "third rail" the mainstream media doesn't want to grab. Yet.

And Paul Barry replies: Yes, and like most sites, they can sometimes take an age to load too. It can be quite instructive to watch my browser's status bar during page

loads (especially on a slow connection).

Even though RTE is our national publicly funded broadcaster, they have always run ads, so no surprise that you're seeing this.

Keep up the good work, *LJ*. BTW, I *love* the podcast too.

Note: check out Doc Searls and Katherine Druckman's weekly Linux Journal *podcast, "Reality 2.0" **here**.—Ed.*

Re: Auto-Download *Linux Journal* Each Month

Things like that [the autolj script] are what make us go back and re-subscribe to this great magazine. Do not lose the geek in you!

—Salahuddin Mohammad ElKazak

If you haven't already heard, subscribers now can download Linux Journal *automatically with our autolj script, which you can get **here**. See this **article** for instructions and more information.—Ed.*

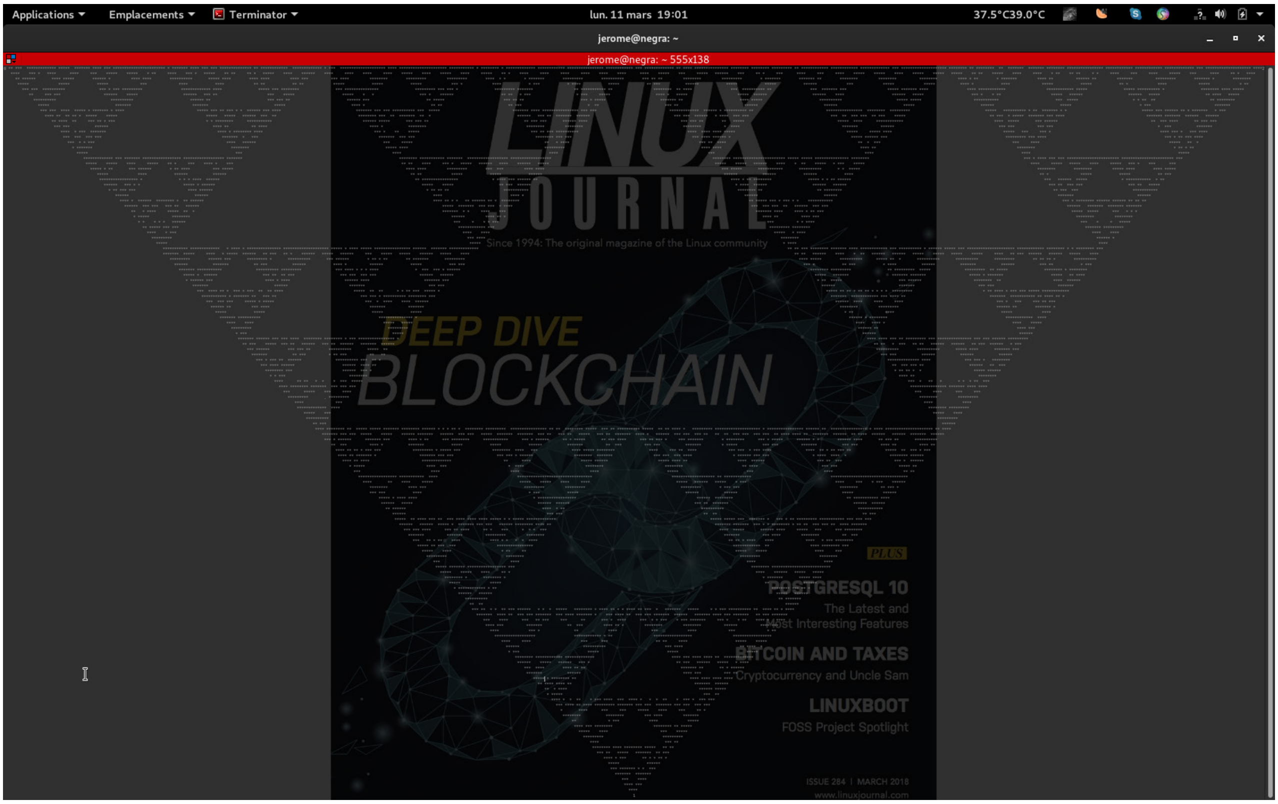
Screenshots from a Longtime Reader

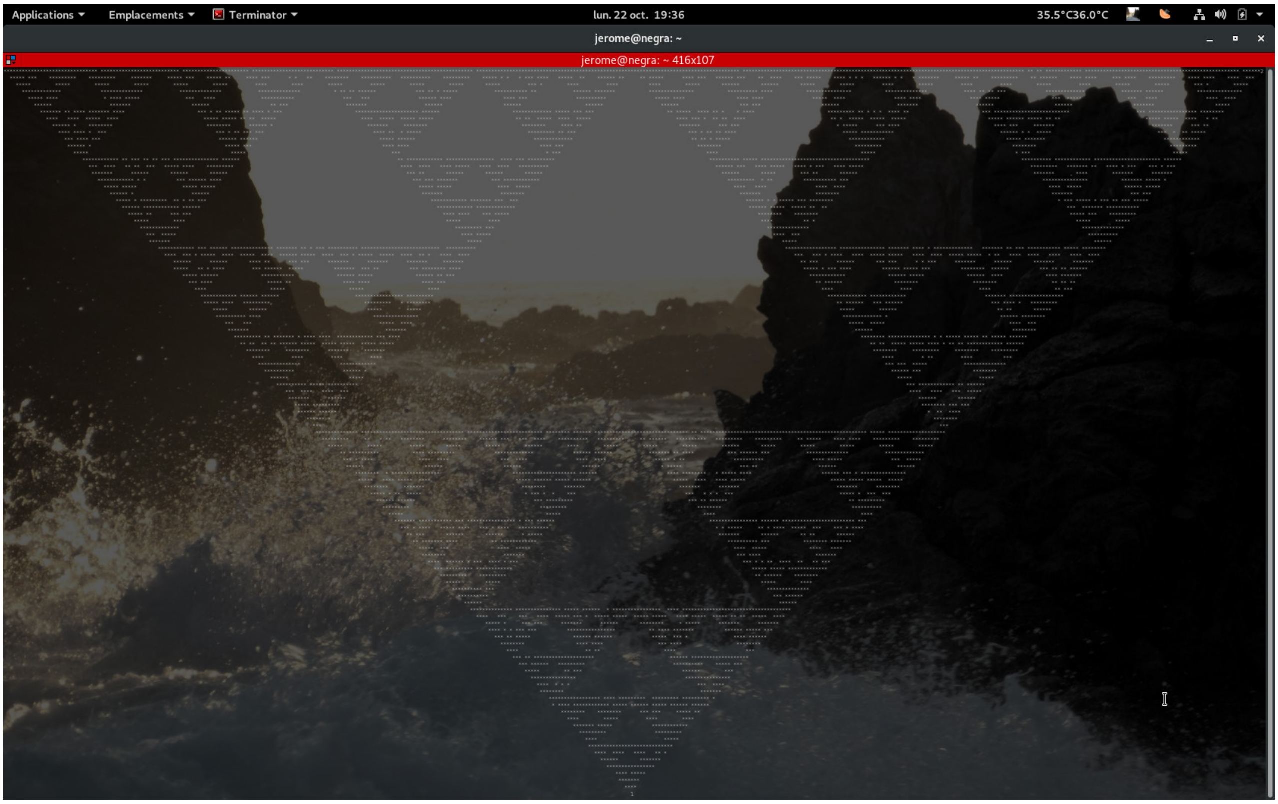
I'm a longtime reader, and I'm most happy that the journal has come back. You're doing a great job.

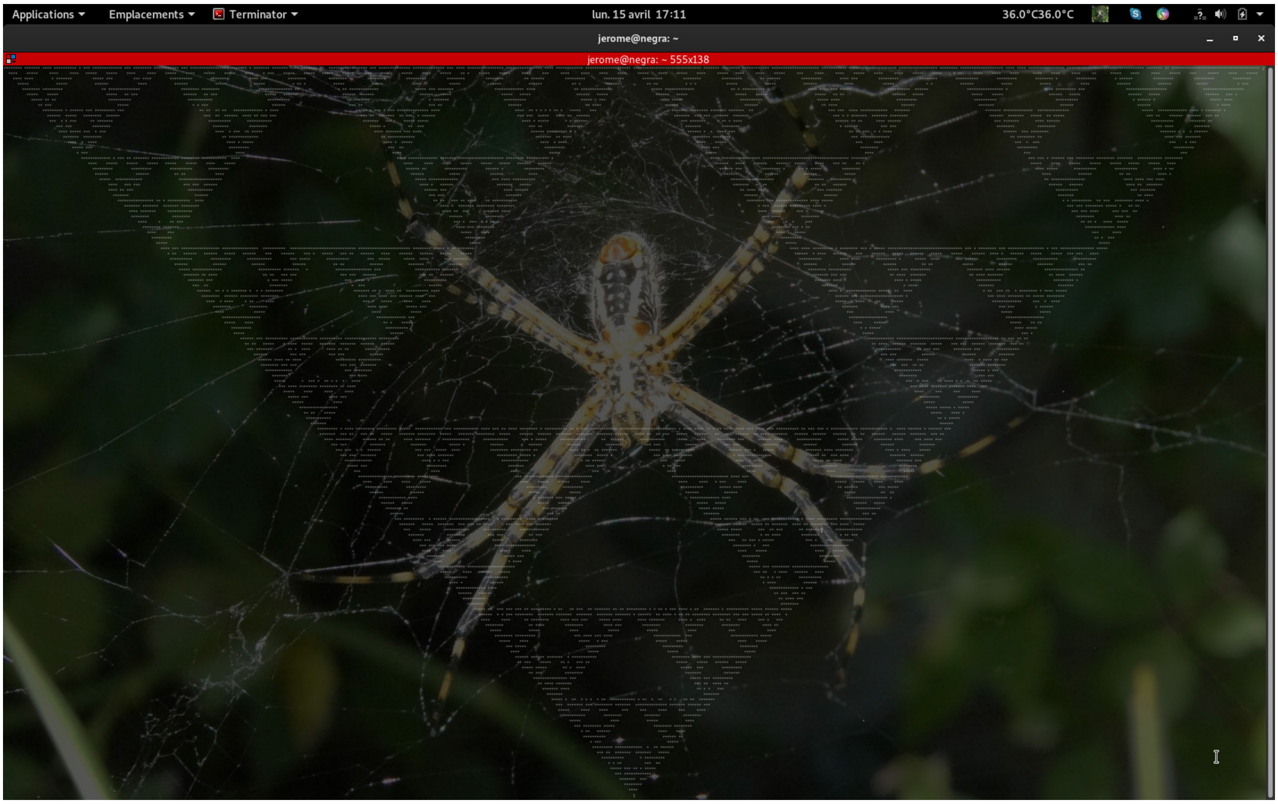
I'm sending some screenshots of my work computer. I use Debian as distribution. I'm a fanatic of mathematics, so I loved the article "Getting Started with ncurses" from Jim Hall in issue #284. The Sierpinski's Triangles are now accompanying me during my work, as I run a full terminal in one screen. All of them are with a background photo I took here in Mexico, and the first one is with the cover of issue #284.

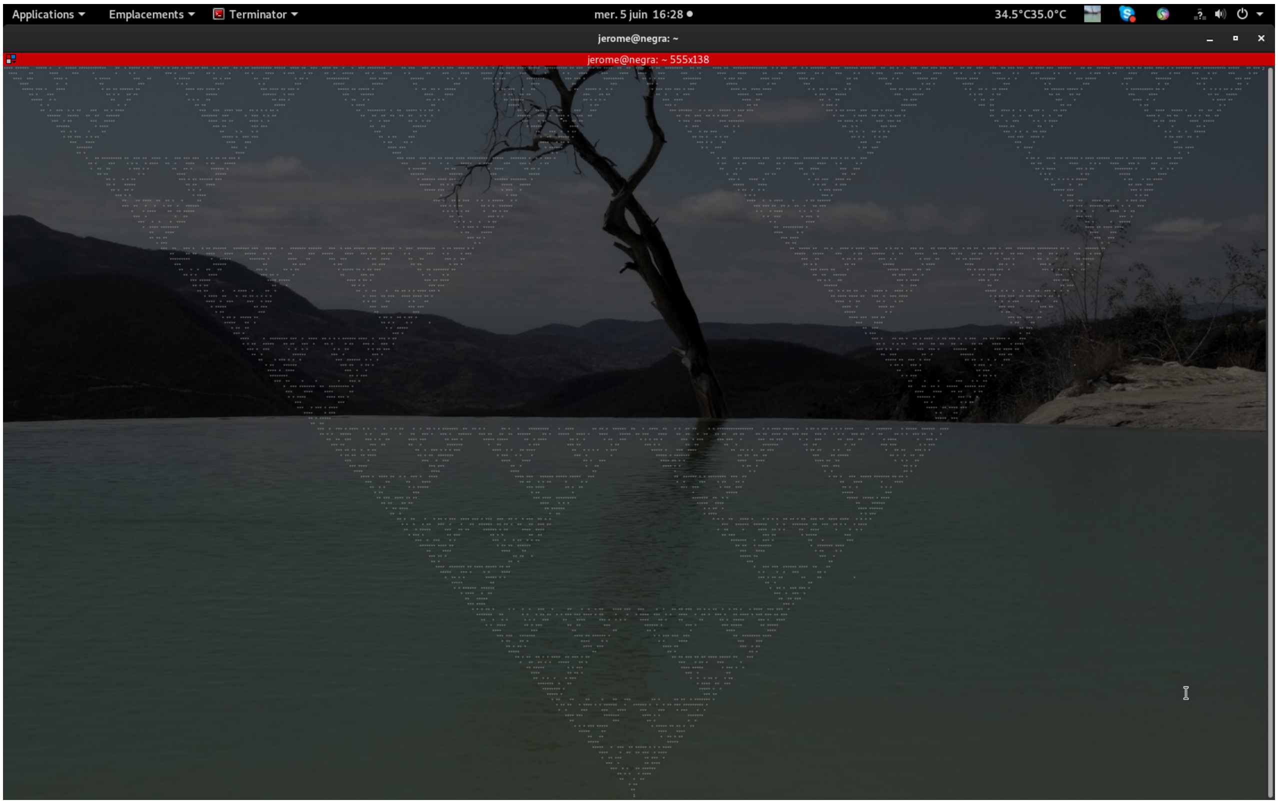
I hope that *LJ* will continue with such great articles!

—Jerome


Applications
Emplacements
Terminator
lun. 11 mars 19:01
37.5°C39.0°C
jerome@negra: ~
jerome@negra: ~ 555x138
LINUX JOURNAL
Since 1994: The original magazine of the Linux community
DEEP DIVE
BLOCKCHAIN
PLUS
POSTGRESQL 10
The Latest and
Most Interesting Features
BITCOIN AND TAXES
Cryptocurrency and Uncle Sam
LINUXBOOT
FOSS Project Spotlight
ISSUE 284 | MARCH 2018
www.linuxjournal.com


Applications
Emplacements
Terminator
lun. 22 oct. 19:36
35.5°C36.0°C
jerome@negra: ~
jerome@negra: ~ 416x107


Applications
Emplacements
Terminator
lun. 15 avril 17:11
36.0°C36.0°C
jerome@negra: ~
jerome@negra: ~ 555x138


Applications
Emplacements
Terminator
mer. 5 juin 16:28
34.5°C35.0°C
jerome@negra: ~
jerome@negra: ~ 555x138

SEND *LJ* A LETTER *We'd love to hear your feedback on the magazine and specific articles. Please write us here or send email to ljeditor@linuxjournal.com.*

PHOTOS *Send your Linux-related photos to ljeditor@linuxjournal.com, and we'll publish the best ones here.*

GIS on Linux with SAGA

In this article, I want to look at a GIS option available for Linux—specifically, a program called SAGA (System for Automated Geoscientific Analyses). SAGA was developed at the Department of Physical Geography in Germany. It is built with a plugin module architecture, where various functions are provided by individual modules. A very complete API is available to allow users to extend SAGA's functionality with newly written modules. I take a very cursory look at SAGA here and describe a few things you might want to do with it.

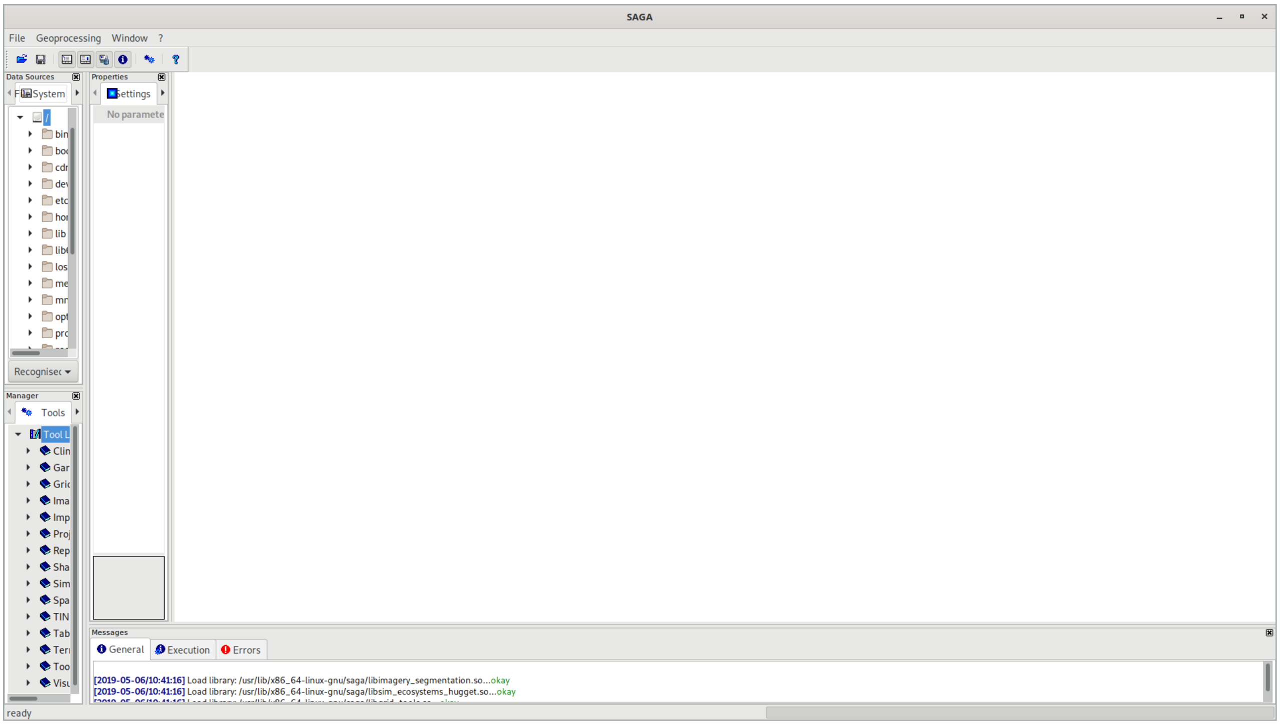


Figure 1. SAGA starts up with a central project window, several tool panes on the left and console messages at the bottom.

Installing SAGA should be as easy as looking at the software repository for your favourite distribution. For Debian-based distros, you can install it with the command:

```
sudo apt-get install saga
```

When you first start it, you get a blank workspace where you can begin your project.

Two major categories of data sets are available that you can use within your projects: satellite imagery and terrain data. The tutorial website provides detailed walk-throughs that show how you can get access to these types of data sets for use in your own projects. The tutorial website also has sections on some of the processing tools available for doing more detailed analysis.

SAGA understands several data file formats. The typical ones used in GIS, like SHP files

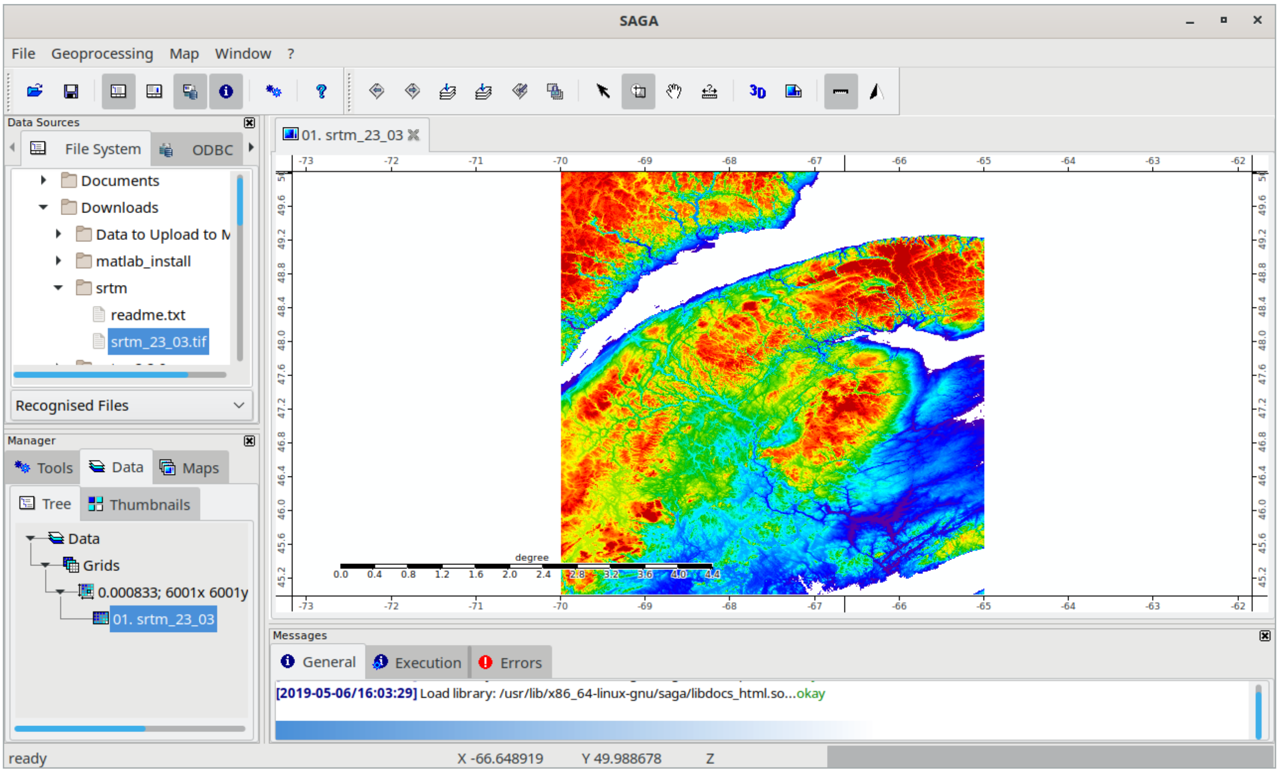


Figure 2. You can load data sources, such as geotiffs, into your project.

or point clouds, are the default options in the file selector window. You can work with these types of data, or satellite imagery or terrain data.

Let's start by looking at terrain analysis in SAGA. You'll need digital elevation data, in DEM format, which is available from the SRTM Tile Grabber site. You will get a zip file for each region you select, and these zip files contain geotiff files for the selected regions.

Load the geotiff file by clicking File→Open. By default, it will show only the common project file formats. To locate your downloaded geotiff files, you'll need to change the filter at the bottom of the file selector window to be all files. Once it is loaded, it will show up in the list of data sources in the bottom-left window pane.

You may find that the default layout is a bit crowded, so you may want to close

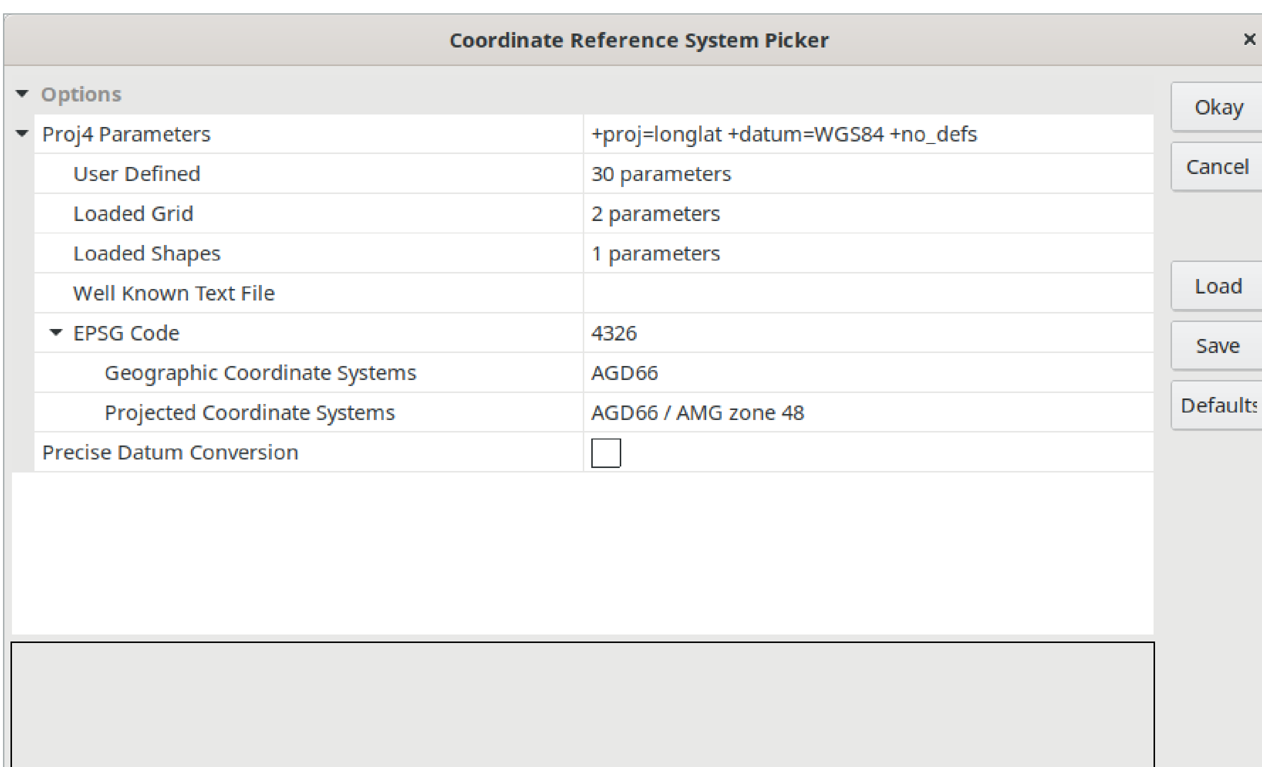


Figure 3. You can play with projection settings for a data source.

some of the detail panes, as I did in the screenshot here. By default, the file is simply added to the list of data sources in the data manager pane in the bottom left of the window, and nothing is displayed. To view the newly added data, you can right-click on the entry in the data manager and select "Add to Map". Several other options are available when you right-click the data source. For example, you can click on "Spatial Reference" to get details about the projection and so on.

Moving to the "Map" tab of the bottom-right pane, you can see the current list of maps and their layers. Right-clicking on the layers provides only viewing options, like what layers lie above or below other layers. Right-clicking on the map provides a bit more functionality. You can save maps as images or even copy them to the clipboard to be used in some other application. You change the overall view to either a 3D view or a print layout. You even can add extra items to your map, such as a base map or a graticule. You also can adjust the order of the layers, in order to have all of the

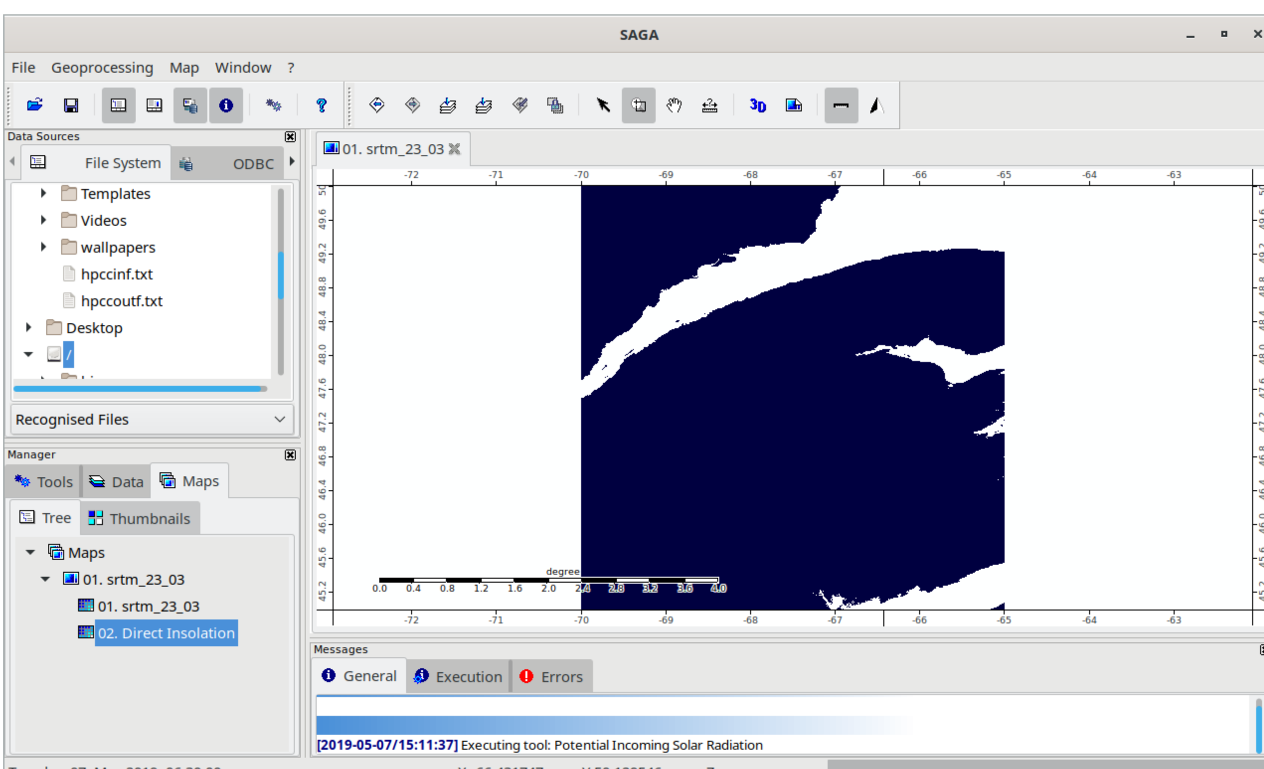


Figure 4. You can run tools that provide additional layers to be displayed in your map.

appropriate information displayed correctly. Several other tools are available under the "map" menu item at the top of the window.

When you click the "Geoprocessing" menu item, you'll see the massive list of tools available to do processing tasks on the data that you have imported. Some of them are basic, while others are very computationally intensive. For example, if you click Terrain Analysis→Basic Terrain Analysis, it can sit and run for quite a while. You will select single analysis tools if you want to look at some specific items. For example, you could look at the solar radiation by clicking Geoprocessing→Terrain Analysis→Lighting→Potential Incoming Solar Radiation.

As you can see, a very dense tree of tools is available. Many of these tools also are available under the "Tools" tab in the bottom-left pane. You simply find the tool in question and double-click it. You may, however, discover that it's difficult to find a

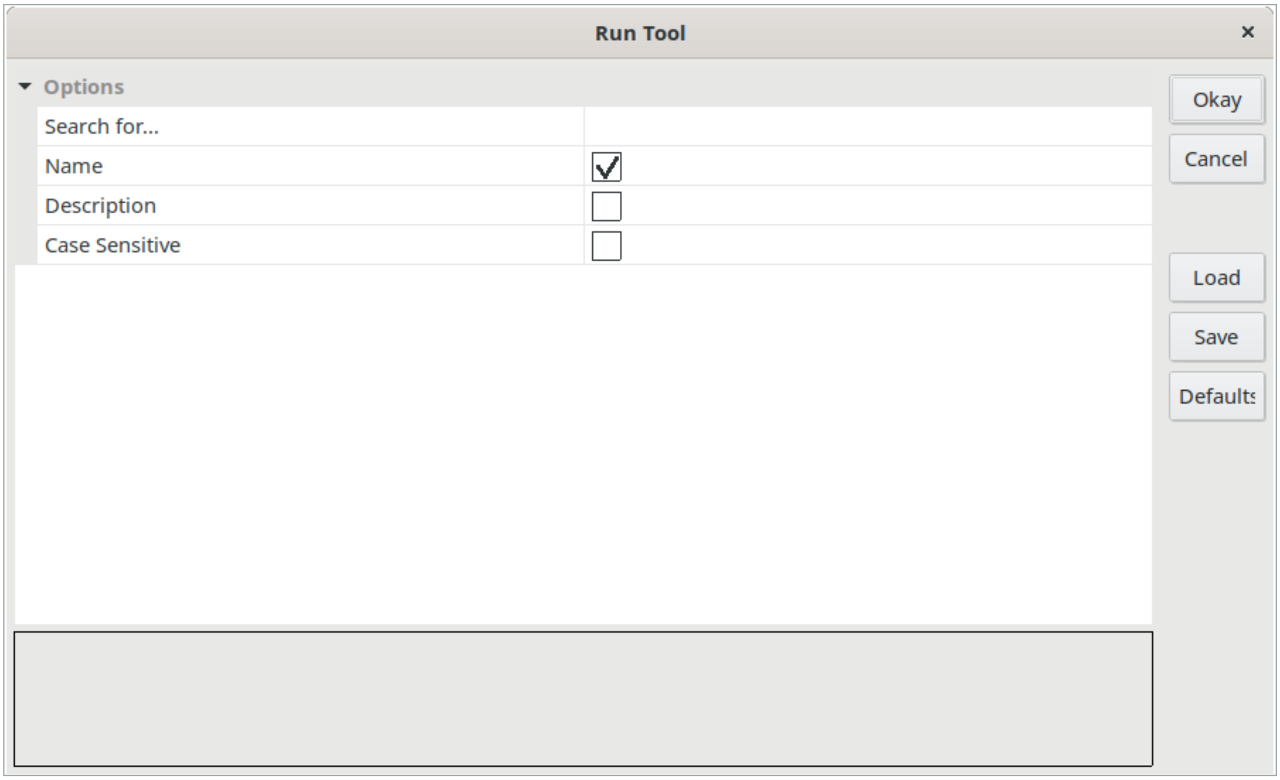


Figure 5. You can do a search for a specific tool, rather than navigating through the menus.

specific tool. If that's the case, click Geoprocessing→Find and Run Tool, and you'll get a pop-up window where you can look for something specific.

To get a better idea of what you can do, the SAGA tutorial site includes a set of complete application examples that walk you through entire workflows. For instance, the first one in the list is assessing sediment flows from a point field survey. It explains how to import data from a CSV file, apply coordinate transformations and visualize the resulting data. It then walks through how to apply a hydrological analysis tool to better understand how the sediment flow happens based on the terrain information.

I hope this short article whets your appetite for using GIS in your own projects. I've covered only a small taste of everything you can do with SAGA, and as with many open-source projects, you always can add extra functionality as needed, which will be loaded as shared libraries that provide additional tools.

—Joey Bernard

Patreon and *Linux Journal*

Together with the help of *Linux Journal* supporters and subscribers, we can offer trusted reporting for the world of open-source today, tomorrow and in the future. To our subscribers, old and new, we sincerely thank you for your continued support. In addition to magazine subscriptions, we are now receiving support from readers via Patreon on our website. *LJ* community members who pledge $20 per month or more will be featured each month in the magazine. A very special thank you this month goes to:

- Appahost.com
- Brian Goodrich
- Chris Short
- Christel Dahlskjaer
- David Breakey
- Dr. Stuart Makowski
- Fred
- Henrik Halbritter (Albritter)
- James Mayes
- Jay M
- Joe
- Josh Simmons
- LinuxMagic Inc.
- Lorin Ricker
- Paul Wood
- Taz Brown

BECOME A PATRON

Now also find @linuxjournal on Liberapay. Thank you to our very first Liberapay supporter and the person who gave us this great suggestion: Mostly_Linux.

Lessons in Vendor Lock-in: Google and Huawei

What happens when you're locked in to a vendor that's too big to fail, but is on the opposite end of a trade war?

The story of Google no longer giving Huawei access to Android updates is still developing, so by the time you read this, the situation may have changed. At the moment, Google has granted Huawei a 90-day window whereby it will have access to Android OS updates, the Google Play store and other Google-owned Android assets. After that point, due to trade negotiations between the US and China, Huawei no longer will have that access.

Whether or not this new policy between Google and Huawei is still in place when this article is published, this article isn't about trade policy or politics. Instead, I'm going to examine this as a new lesson in vendor lock-in that I don't think many have considered before: what happens when the vendor you rely on is forced by its government to stop you from being a customer?

Too Big to Fail

Vendor lock-in isn't new, but until the last decade or so, it generally was thought of by engineers as a bad thing. Companies would take advantage the fact that you used one of their products that was legitimately good to use the rest of their products that may or may not be as good as those from their competitors. People felt the pain of being stuck with inferior products and rebelled.

These days, a lot of engineers have entered the industry in a world where the new

giants of lock-in are still growing and have only flexed their lock-in powers a bit. Many engineers shrug off worries about choosing a solution that requires you to use only products from one vendor, in particular if that vendor is a large enough company. There is an assumption that those companies are too big ever to fail, so why would it matter that you rely on them (as many companies in the cloud do) for every aspect of their technology stack?

Many people who justify lock-in with companies who are too big to fail point to all of the even more important companies who use that vendor who would have even bigger problems should that vendor have a major bug, outage or go out of business. It would take so much effort to use cross-platform technologies, the thinking goes, when the risk of going all-in with a single vendor seems so small.

Huawei also probably figured (rightly) that Google and Android were too big to fail. Why worry about the risks of being beholden to a single vendor for your OS when that vendor was used by other large companies and would have even bigger problems if the vendor went away?

The Power of Updates

Google held a particularly interesting and subtle bit of lock-in power over Huawei (and any phone manufacturer who uses Android)—the power of software updates. This form of lock-in isn't new. Microsoft famously used the fact that software updates in Microsoft Office cost money (naturally, as it was selling that software) along with the fact that new versions of Office had this tendency to break backward compatibility with older document formats to encourage everyone to upgrade. The common scenario was that the upper-level folks in the office would get brand-new, cutting-edge computers with the latest version of Office on them. They would start saving new documents and sharing them, and everyone else wouldn't be able to open them. It ended up being easier to upgrade everyone's version of Office than to have the bosses remember to save new documents in old formats every time.

The main difference with Android is that updates are critical not because of compatibility, but for security. Without OS updates, your phone ultimately will

become vulnerable to exploits that attackers continue to find in your software. The Android OS that ships on phones is proprietary and therefore requires permission from Google to get those updates.

Many people still don't think of the Android OS as proprietary software. Although people talk about the FOSS underpinnings in Android, only people who go to the extra effort of getting a pure-FOSS version of Android, like LineageOS, on their phones actually experience it. The version of Android most people tend to use has a bit of FOSS in the center, surrounded by proprietary Google Apps code.

It's this Google Apps code that gives Google the kind of powerful leverage over a company like Huawei. With traditional Android releases, Google controls access to OS updates including security updates. All of this software is signed with Google's signing keys. This system is built with security in mind—attackers can't easily build their own OS update to install on your phone—but it also has a convenient side effect of giving Google control over the updates.

What's more, the Google Apps suite isn't just a convenient way to load Gmail or Google Docs, it also includes the tight integration with your Google account and the Google Play store. Without those hooks, you don't have access to the giant library of applications that everyone expects to use on their phones. As anyone with a LineageOS phone that uses F-Droid can attest, while a large number of applications are available in the F-Droid market, you can't expect to see those same apps as on Google Play. Although you can side-load some Google Play apps, many applications, such as Google Maps, behave differently without a Google account. Note that this control isn't unique to Google. Apple uses similar code-signing features with similar restrictions on its own phones and app updates.

Conclusion

Without access to these OS updates, Huawei now will have to decide whether to create its own LineageOS-style Android fork or a whole new phone OS of its own. In either case, it will have to abandon the Google Play Store ecosystem and use F-Droid-style app repositories, or if it goes 100% alone, it will need to create a completely new

app ecosystem. If its engineers planned for this situation, then they likely are working on this plan right now; otherwise, they are all presumably scrambling to address an event that "should never happen". Here's hoping that if you find yourself in a similar case of vendor lock-in with an overseas company that's too big to fail, you never get caught in the middle of a trade war.

—Kyle Rankin

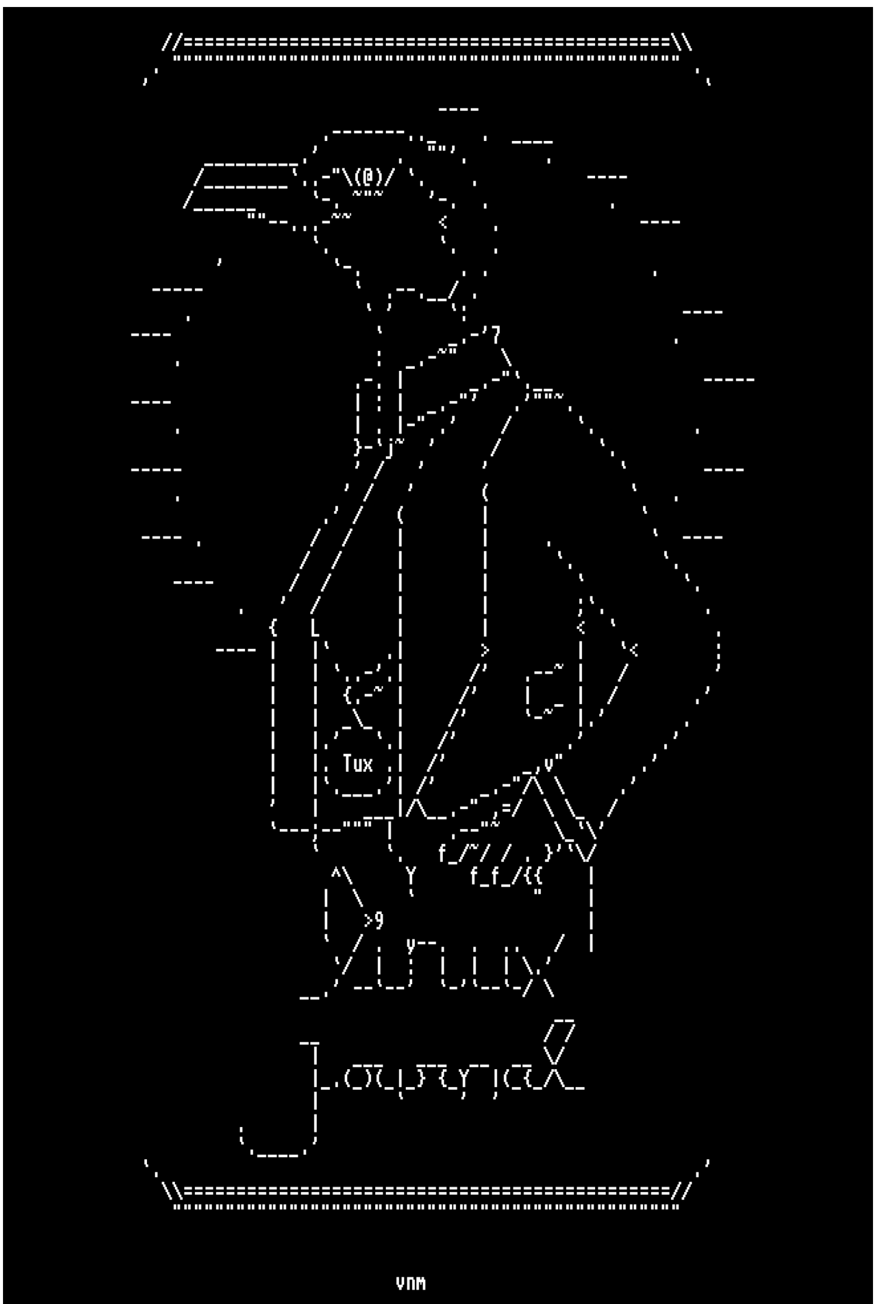
Reality 2.0: a *Linux Journal* Podcast

Join us each week as Doc Searls and Katherine Druckman navigate the realities of the new digital world: https://www.linuxjournal.com/podcast.

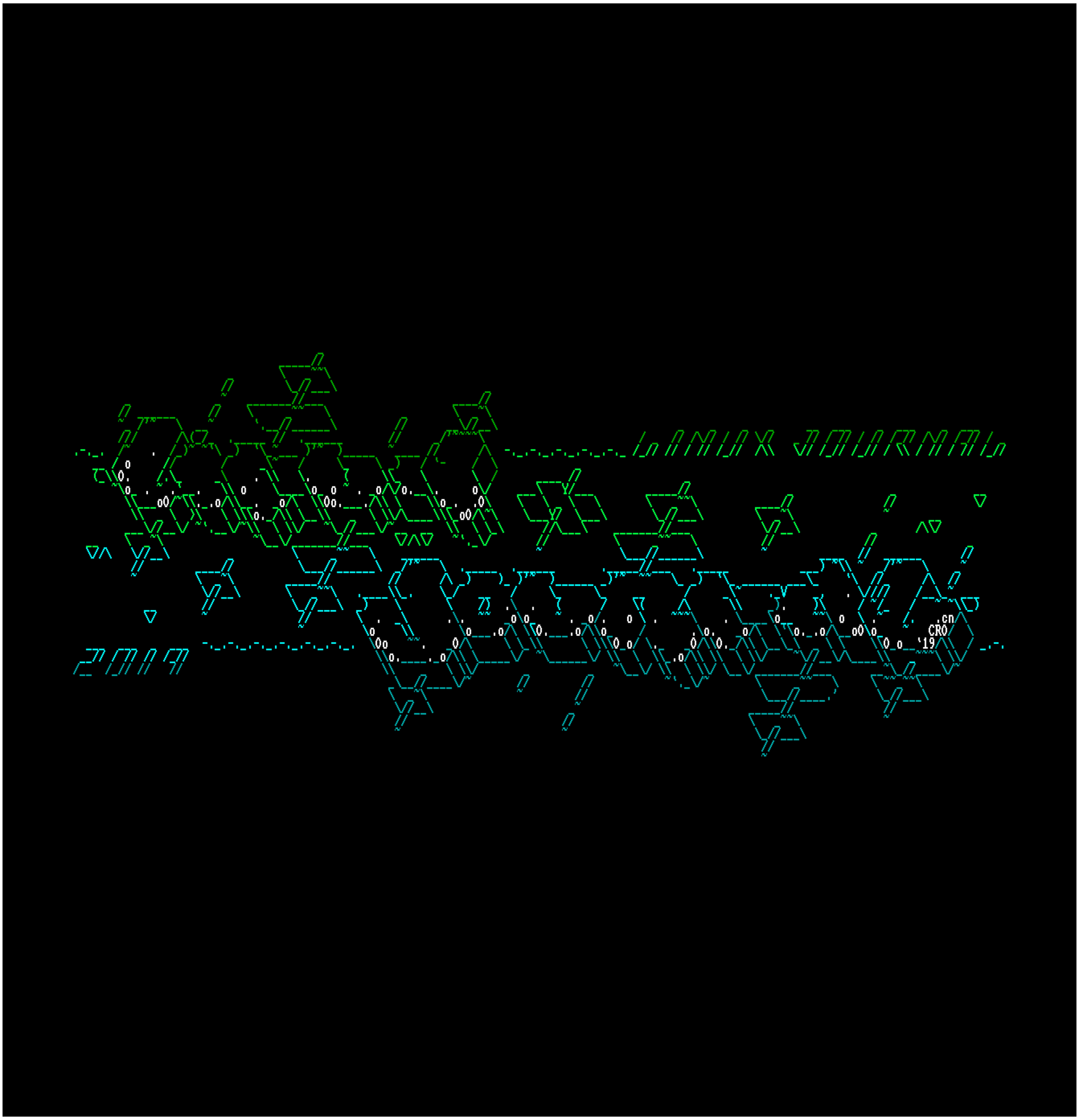


ASCII Art Contest

The image on this month's cover by aNACHRONiST (Daniel Kelly) is the winner of our contest. Below are the runners-up. Thanks to all who participated!



By Patrick Louis, aka venam or vnm



By .cn/CRO

News Briefs

Visit LinuxJournal.com for daily news briefs.

- Windows and Chrome are making 2019 the "year of the desktop". PCWorld writes, "After years of endless jokes, 2019 is truly, finally shaping up to be the year of Linux on the desktop. Laptops, too! But most people won't know it. That's because the bones of the open-source operating system kernel will soon be baked into Windows 10 and Chrome OS, as Microsoft and Google revealed at their respective developer conferences this week."

- Schools in the Indian state of Kerala have chosen Linux as their OS, which will save them roughly $428 million. According to It's FOSS, Kerala is "the first 100% literate Indian state". IT classes have been mandatory since 2003, and the schools started adopting free and open-source software a few years later, with the plan of getting rid of proprietary software in the schools. "As a result, the state claimed to save around $50 million per year in licensing costs in 2015. Further expanding their open source mission, Kerala is going to put Linux with open source educational software on over 200,000 school computers."

- Nextcloud announced a new partnership with Nitrokey, maker of highly secure, open-source encryption USB keys. From the press release: "The Nitrokey Pro 2 and Nitrokey Storage 2 devices have been verified to work easily with Nextcloud's one-time passwords for secure two-factor authentication (2FA). This protects users' accounts in the event of compromised passwords. Furthermore the USB keys feature a password manager, a cryptographic key store for email encryption and SSH administration. In addition the Nitrokey Storage 2 contains an encryption mass storage drive with the option of hidden volumes." Nextcloud and Nitrokey also will explore further collaboration "especially in the area of end-to-end encryption and secure storage of cryptographic keys". See the Nextcloud blog for more details.

- The Linux Foundation announced the formation of the Urban Computing Foundation "to accelerate open source software that improves mobility,

safety, road infrastructure, traffic congestion and energy consumption in connected cities. Initial contributors include developers from Uber, Facebook, Google, HERE Technologies, IBM, Interline Technologies, Senseable City Labs, StreetCred Labs and University of California San Diego (UCSD)." The Foundation's first project is kepler.gl, "an open-source geospatial analysis tool created by Uber for building large-scale data sets".

- The Atomic Pi has recently hit retail channels after its successful Kickstarter campaign (although it was sold out at the time of this writing). Phoronix reports that the $35 Atomic Pi "offers an Intel Atom x5-Z8350 quad-core, 2GB DDR3L-1600 memory, 16GB eMMC, SD slot, USB 3.0/2.0 ports, 802.11ac WiFI, Bluetooth 4.0, and Gigabit Ethernet". The article also notes that "It's quite a board for the price and [will] compete with the likes of the Raspberry Pi." Go to Digital Loggers for more information.

- Hewlett Packard Enterprise is buying supercomputer-maker Cray. Bloomberg reports that the deal is "valued at about $1.4 billion as the firm works to become more competitive in high-end computing", and that "Cray investors will get $35 a share in cash".

- Researchers have discovered another Intel processor vulnerability called Zombieload. According to ZDNet, "The researchers have shown a Zombieload exploit that can look over your virtual shoulder to see the websites you're visiting in real-time. Their example showed someone spying on another someone using the privacy-protecting Tor Browser running inside a virtual machine (VM)." But there's some good news: "To defend yourself, your processor must be updated, your operating system must be patched, and for the most protection, Hyper-Threading disabled. When Meltdown and Spectre showed up, the Linux developers were left in the dark and scrambled to patch Linux. This time, they've been kept in the loop."

- The Antergos Linux distro is calling it quits. The developers of the Arch-based distro say they no longer have time to maintain it properly, and they are taking

action now while the code is still working in case other developers want to start their own projects with it. From the Antergos blog: "For existing Antergos users: there is no need to worry about your installed systems as they will continue to receive updates directly from Arch. Soon, we will release an update that will remove the Antergos repos from your system along with any Antergos-specific packages that no longer serve a purpose due to the project ending. Once that is completed, any packages installed from the Antergos repo that are in the AUR will begin to receive updates from there."

- GitHub launched a new tool called Sponsors that lets you make payments to open-source developers. Tech Crunch reports that "Developers will be able to opt into having a 'Sponsor me' button on their GitHub repositories and open source projects will also be able to highlight their funding models, no matter whether that's individual contributions to developers or using Patreon, Tidelift, Ko-fi or Open Collective.

- Feral Interactive announced that *Total War: THREE KINGDOMS* is out on Linux and macOS (the same day as the Windows release). The game was developed by Creative Assembly and is the first in the *Total War* series to be set in ancient China. It's available now from the Feral Interactive Store for $59.99, and you can watch the trailer here.

- You can send the E Foundation your phone if you'd like a Google-free Android. FOSS Bytes reports that with the E Foundation's /e/ OS, "the main goal of /e/ is to take away Google's control over the device. It doesn't include any Google apps that you'd normally find on Android phones. Other than UI tweaks and pre-loading all the essential apps like Browser, Contacts, Calendar, Messaging, it even has an App Store of its own. You can also have an /e/ account, and take advantage of its cloud storage service, mail, and search." The E Foundation will soon be selling refurbished devices with the OS here, and according to Foss Bytes, you will be able to send them your phone, and they will install it for around $50. Or, you can flash your phone yourself and install the beta ROM, which you can download from here. It currently supports 81 devices from

Google, Motorola, Huawei, Samsung and more.

- Raspberry Pi Camera Modules mounted on Raspberry Pi Zeros provide the images for the **Penguin Watch** project. The **raspberrypi.org blog post** calls the project "citizen science on a big scale", noting that "thousands of people from all over the world come together on the internet to...click on penguins. By counting the birds in their colonies, users help penguinologists measure changes in the birds' behaviour and habitat, and in the larger ecosystem, thus assisting in their conservation."

- System76 announced the rebirth of its Gazelle laptop line, offering the choice of Pop!_OS or Ubuntu as the OS. **Beta News reports**, "It comes with a 9th Gen Intel Core i7 by default, and you can choose between an NVIDIA GeForce GTX 1650 or 1660 Ti for graphics. There are two screen sizes available -- 15.3-inch and 17.3-inch. Regardless of the display you opt for, the resolution will be 1080p." See the full specs and sign up to be notified when the laptops are available (which should be sometime before this is published) **here**.

- Mozilla announced that the Firefox browser will now have Enhanced Tracking Protection on by default. From **Chris Beard's blog post**: "These protections work in the background, blocking third-parties from tracking your online activity while increasing the speed of the browser. We're offering privacy protections by default as you navigate the web because the business model of the web is broken, with more and more intrusive personal surveillance becoming the norm. While we hope that people's digital rights and freedoms will ultimately be guaranteed, we're here to help in the interim."

Send comments or feedback
via https://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

What Really IRCs Me: Mastodon



Kyle Rankin is a Tech Editor and columnist at *Linux Journal* and the Chief Security Officer at Purism. He is the author of *Linux Hardening in Hostile Networks*, *DevOps Troubleshooting*, *The Official Ubuntu Server Book*, *Knoppix Hacks*, *Knoppix Pocket Reference*, *Linux Multimedia Hacks* and *Ubuntu Hacks*, and also a contributor to a number of other O'Reilly books. Rankin speaks frequently on security and open-source software including at BsidesLV, O'Reilly Security Conference, OSCON, SCALE, CactusCon, Linux World Expo and Penguicon. You can follow him at @kylerankin.

Learn how to use the Mastodon social network platform from the comfort of your regular IRC client.

By Kyle Rankin

When it comes to sending text between people, I've found IRC (in particular, a text-based IRC client) works best. I've been using it to chat for decades while other chat protocols and clients come and go. When my friends have picked other chat clients through the years, I've used the amazing IRC gateway Bitlbee to connect with them on their chat client using the same IRC interface I've always used. Bitlbee provides an IRC gateway to many different chat protocols, so you can connect to Bitlbee using your IRC client, and it will handle any translation necessary to connect you to the remote chat clients it supports. I've written about Bitlbee a number of times in the past, and I've used it to connect to other instant messengers, Twitter and Slack. In this article, I describe how I use it to connect to yet another service on the internet: Mastodon.

Like Twitter, Mastodon is a social network platform, but unlike Twitter, Mastodon runs on free software and is decentralized, much like IRC or email. Being decentralized means it works similar to email, and you can create your own instance or create

an account on any number of existing Mastodon networks and then follow people either on the same Mastodon network or any other instance, as long as you know the person's user name (which behaves much like an email address).

I've found Bitlbee to be a great interface for keeping track of social media on Twitter, because I treat reading Twitter like I was the operator for a specific IRC room. The people I follow are like those I've invited and given voice to, and I can read what they say chronologically in my IRC room. Since I keep my IRC instance running at all times, I can reconnect to it and catch up with the backlog whenever I want. Since I'm reading Twitter over a purely text-based IRC client, this does mean that instead of animated gifs, I just see URLs that point to the image, but honestly, I consider that a feature!

Since Mastodon behaves in many ways like Twitter, using it with Bitlbee works just as well. Like with Twitter over Bitlbee, it does mean you'll need to learn some extra commands so that you can perform Mastodon-specific functions, like boosting a post (Mastodon's version of retweet) or replying to a post so that your comment goes into the proper thread. I'll cover those commands in a bit.

Installing the Mastodon Bitlbee Plugin

The first step is to install the Mastodon Bitlbee Plugin. This plugin is already packaged for Debian and other distributions—look for the `bitlbee-mastodon` package. In that case, you can just install it with your package manager. Otherwise, you'll need to clone the source code from the plugin's git repo and build it from source:

```
git clone https://alexschroeder.ch/cgit/bitlbee-mastodon
cd bitlbee-mastodon
./autogen.sh
./configure
make
sudo make install
```

Note that if you build it from source, you need to have the Bitlbee development package installed on your distribution (usually called `bitlbee-dev` or `bitlbee-devel`).

Configure Mastodon

Once you have installed the plugin, restart the bitlbee service (`sudo service bitlbee restart` should work on most distributions these days). Then when you connect to Bitlbee with your IRC client, make sure you are in the specific &bitlbee IRC channel it creates. From that room, you'll be able to register your new account using the standard account tools within Bitlbee. For this example, let's assume I want to connect to my @kyle@librem.one Mastodon account:

```
@greenfly| account add mastodon @kyle
@    root| Account successfully added with tag mastodon
```

This creates a new Mastodon account in Bitlbee and names it "mastodon", but note that if I already had a Mastodon account present, it would have started adding numbers at the end, such as "mastodon2" instead. Pay attention to this tag, as you will use it in the next command to configure this Mastodon account to point it at your particular Mastodon network:

```
@greenfly| account mastodon set base_url
 ↪https://social.librem.one/api/v1
@    root| base_url = 'https://social.librem.one/api/v1'
@greenfly| account mastodon on
```

At this point, Bitlbee will connect to Mastodon, and you'll need to authenticate this client. You'll get a private message from the mastodon_oauth user that will send you a URL to visit in a browser. When you visit the URL, you'll see a long string of text that you'll need to copy and then paste back as a reply to the mastodon_oauth user:

```
mastodon_oauth| Open this URL in your browser to authenticate:
https://social.librem.one/...
mastodon_oauth| Respond to this message with the returned
 ↪authorization token.
      greenfly| somelongstringoftext
```

After you complete this authentication step, you can go back to the main &bitlbee channel, and you'll see that your login has completed. From this point on, your Bitlbee Mastodon account will have an authentication token it can use to log in in the future, so in that &bitlbee window, be sure to save your configuration:

```
@greenfly| save
@   root | Configuration saved
```

Using Mastodon

If you are familiar with using Twitter on Bitlbee, using Mastodon is similar. Bitlbee will open a new IRC channel for your Mastodon account, and anything anyone posts will show up there. Anything you type into the channel will be posted on your Mastodon account by default. If you want to restrict that so it posts things only when you explicitly use the `post` command, you'll need to set your Mastodon account to strict mode, so inside the main &bitlbee control channel, type:

```
@greenfly| account mastodon set commands strict
```

To revert to the default behavior, type:

```
@greenfly| account mastodon set commands true
```

To post something, either just type the message and press Enter, like any other IRC channel, or if you have enabled strict commands, preface the post with the `post` command:

```
post My first toot from Bitlbee!
```

Each post in your Mastodon channel will be prefaced with a hexadecimal ID. You can use that ID if you want to boost, favorite or reply to that status. For instance, if you saw a post like the following:

```
kyle| [15] My first toot from Bitlbee!
```

You would use the ID "15" to interact with it:

```
favorite 15
boost 15
reply 15 The Mastodon Bitlbee plugin is the best!
```

If you decide you want to remove that last reply, you can use either the `undo` command to undo your last action or the `del` command to delete a particular status based on ID. You can also `unfavorite` and `unboost` to undo those commands.

To follow an account, use the `follow` command along with the person's Mastodon account ID (for instance @kyle@librem.one), and use `unfollow` along with the account ID to unfollow someone. You can block, unblock, mute and unmute someone using the commands with the same names. You also can mute a particular conversation by specifying the post's ID. Finally, you can use the `report` command to a particular status for moderation.

Searches

You can perform Mastodon searches and create whole rooms that follow hashtags from within Bitlbee as well. The search command lets you search for users, hashtags and the URL for a particular status. If you want to follow a hashtag in its own room, you need to perform a series of Bitlbee commands to create the chat room. For instance, if you wanted to follow the hashtag #linuxjournal you would type:

```
chat add mastodon hashtag #linuxjournal
channel #linuxjournal set auto_join true
/join #linuxjournal
```

The first command sets that hashtag search in the account named "mastodon", but if you have multiple Mastodon accounts, replace "mastodon" with "mastodon2" or whatever the appropriate account is labeled. The second command creates the specific channel related to that hashtag search, and the `/join` command will connect to that channel.

Conclusion

Reading social media from IRC makes following lots of people much easier and feels more like a regular chat room. Although it's true that you can't immediately see images, some people probably would consider that as a feature. Even better, it means you can channel yet another communication medium through IRC and not have to learn the quirks of a new client. ■

Send comments or feedback via http://www.linuxjournal.com/contact or email ljeditor@linuxjournal.com.

Python's Mypy: Callables and Generators



Reuven Lerner teaches Python, data science and Git to companies around the world. You can subscribe to his free, weekly "better developers" e-mail list, and learn from his books and courses at http://lerner.co.il. Reuven lives with his wife and children in Modi'in, Israel.

Learn how Mypy's type checking works with functions and generators.

By Reuven M. Lerner

In my last two articles I've described some of the ways Mypy, a type checker for Python, can help identify potential problems with your code. [See "Introducing Mypy, an Experimental Optional Static Type Checker for Python" and "Python's Mypy—Advanced Usage".] For people (like me) who have enjoyed dynamic languages for a long time, Mypy might seem like a step backward. But given the many mission-critical projects being written in Python, often by large teams with limited communication and Python experience, some kind of type checking is an increasingly necessary evil.

It's important to remember that Python, the language, isn't changing, and it isn't becoming statically typed. Mypy is a separate program, running outside Python, typically as part of a continuous integration (CI) system or invoked as part of a Git commit hook. The idea is that Mypy runs before you put your code into production, identifying where the data doesn't match the annotations you've made to your variables and function parameters.

I'm going to focus on a few of Mypy's advanced features here. You might not encounter them very often, but even if you don't, it'll give you a better picture of the complexities associated with type checking, and how deeply the Mypy team is thinking about their work, and what tests need to be done. It'll also help you understand more about the ways people do type checking, and how to balance the beauty, flexibility and expressiveness of dynamic typing with the strictness and fewer errors of static typing.

Callable Types

When I tell participants in my Python classes that everything in Python is an object, they nod their heads, clearly thinking, "I've heard this before about other languages." But then I show them that functions and classes are both objects, and they realize that Python's notion of "everything" is a bit more expansive than theirs. (And yes, Python's definition of "everything" isn't as wide as Smalltalk's.)

When you define a function, you're creating a new object, one of type "function":

```
>>> def foo():
...     return "I'm foo!"

>>> type(foo)
<class 'function'>
```

Similarly, when you create a new class, you're adding a new object type to Python:

```
>>> class Foo():
...     pass

>>> type(Foo)
<class 'type'>
```

It's a pretty common paradigm in Python to write a function that, when it runs, defines and runs an inner function. This is also known as a "closure", and it has a few

different uses. For example, you can write:

```
def foo(x):
   def bar(y):
       return f"In bar, {x} * {y} = {x*y}"
   return bar
```

You then can run:

```
b = foo(10)
print(b(2))
```

And you'll get the following output:

```
In bar, 10 * 2 = 20
```

I don't want to dwell on how all of this works, including inner functions and Python's scoping rules. I do, however, want to ask the question "how can you use Mypy to check all of this?"

You could annotate both `x` and `y` as `int`. And you can annotate the return value from `bar` as a string. But how can you annotate the return value from `foo`? Given that, as shown above, functions are of type `function`, perhaps you can use that. But `function` isn't actually a recognized name in Python.

Instead, you'll need to use the `typing` module, which comes with Python 3 so you can do this kind of type checking. And in `typing`, the name `Callable` is defined for precisely this purpose. So you can write:

```
from typing import Callable


def foo(x: int) -> Callable:
```

```
    def bar(y: int) -> str:
        return f"In bar, {x} * {y} = {x*y}"
    return bar


b = foo(10)
print(b(2))
```

Sure enough, this passes Mypy's checks. The function `foo` returns `Callable`, a description that includes both functions and classes.

But, wait a second. Maybe you don't only want to check that `foo` returns a `Callable`. Maybe you also want to make sure that it returns a function that takes an `int` as an argument. To do that, you'll use square brackets after the word `Callable`, putting two elements in those brackets. The first will be a list (in this case, a one-element list) of argument types. The second element in the list will describe the return type from the function. In other words, the code now will look like this:

```
#!/usr/bin/env python3

def foo(x: int) -> Callable[[int], str]:
    def bar(y: int) -> str:
        return f"In bar, {x} * {y} = {x*y}"
    return bar


b = foo(10)
print(b(2))
```

Generators

With all this talk of callables, you also should consider what happens with generator functions. Python loves iteration and encourages you to use `for` loops wherever you can. In many cases, it's easiest to express your iterator in the form of a function,

known in the Python world as a "generator function". For example, you can create a generator function that returns the Fibonacci sequence as follows:

```
def fib():
   first = 0
   second = 1
   while True:
       yield first
       first, second = second, first+second
```

You then can get the first 50 Fibonacci numbers as follows:

```
g = fib()
for i in range(50):
   print(next(g))
```

That's great, but what if you want to add Mypy checking to your `fib` function? It would seem that you can just say that the return value is an integer:

```
def fib() -> int:
   first = 0
   second = 1
   while True:
       yield first
       first, second = second, first+second
```

But if you try running this via Mypy, you get a pretty stern response:

```
atf201906b.py:4: error: The return type of a generator function
should be "Generator" or one of its supertypes
atf201906b.py:14: error: No overload variant of "next" matches
argument type "int"
atf201906b.py:14: note: Possible overload variant:
```

```
atf201906b.py:14: note:   def [_T] next(i: Iterator[_T]) -> _T
atf201906b.py:14: note:   <1 more non-matching overload not
shown>
```

Whoa! What's going on?

Well, it's important to remember that the result of running a generator function is *not* whatever you're yielding with each iteration. Rather, the result is a generator object. The generator object, in turn, then yields a particular type with each iteration.

So what you really want to do is tell Mypy that `fib` will return a generator, and that with each iteration of the generator, you'll get an integer. You would think that you could do it this way:

```
from typing import Generator

def fib() -> Generator[int]:
   first = 0
   second = 1
   while True:
       yield first
       first, second = second, first+second
```

But if you try to run Mypy, you get the following:

```
atf201906b.py:6: error: "Generator" expects 3 type arguments,
but 1 given
```

It turns out that the Generator type can (optionally) get arguments in square brackets. But if you provide any arguments, you must provide three:

- The type returned with each iteration—what you normally think about from iterators.

- The type that the generator will *receive*, if you invoke the `send` method on it.
- The type that will be returned when the generator exits altogether.

Since only the first of these is relevant in this program, you'll pass `None` for each of the other values:

```
from typing import Generator

def fib() -> Generator[int, None, None]:
   first = 0
   second = 1
   while True:
       yield first
       first, second = second, first+second
```

Sure enough, it now passes Mypy's tests.

Conclusion

You might think that Mypy isn't up to the task of dealing with complex typing problems, but it actually has been thought out rather well. And of course, what I've shown here (and in my previous two articles on Mypy) is just the beginning; the Mypy authors have solved all sorts of problems, from modules mutually referencing each others' types to aliasing long type descriptions.

If you're thinking of tightening up your organization's code, adding type checking via Mypy is a great way to go. A growing number of organizations are adding its checks, little by little, and are enjoying something that dynamic-language advocates have long ignored, namely that if the computer can check what types you're using, your programs actually might run more smoothly. ■

Resources

You can read more about Mypy here. That site has documentation, tutorials and even information for people using Python 2 who want to introduce `mypy` via comments (rather than annotations).

You can read more about the origins of type annotations in Python, and how to use them, in PEP (Python enhancement proposal) 484, available online here.

See my previous two articles on Mypy: "Introducing Mypy, an Experimantal Optional Static Type Cheker for Python" and "Python's Mypy—Advanced Usage".

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

Bash Shell Games: Let's Play *Go Fish!*

How to begin developing a computer version of the popular card game.

By Dave Taylor



Dave Taylor has been hacking shell scripts on Unix and Linux systems for a really long time. He's the author of *Learning Unix for Mac OS X* and *Wicked Cool Shell Scripts*. You can find him on Twitter as @DaveTaylor, and you can reach him through his tech Q&A site Ask Dave Taylor.

Between the previous 163 columns I've written here in *Linux Journal* and the dozens of games I programmed and explored during the creation of my *Wicked Cool Shell Scripts* book, I've written a lot of Bash shell games. The challenge is to find one that's simple enough where a shell script will work, but isn't so simple that it ends up being only a half-dozen lines.

Magic 8-Ball is a perfect example. It turns out that the entire "predict the future" gizmo was really just a 20-sided die floating in dark purple fluid. So an array of 20 possible values and a random number selector and boom—you've got a magic 8-ball script:

```
#!/bin/sh

# magic 8 ball. Yup. Pick a random number, output message

# messages harvested from the Wikipedia entry
```

```
answers=("It is certain." "It is decidedly so."
  "Without a doubt." "Yes - definitely."
  "You may rely on it." "As I see it, yes." "Most likely."
  "Outlook good." "Yes." "Signs point to yes."
  "Reply hazy, try again." "Ask again later."
  "Better not tell you now." "Cannot predict now."
  "Concentrate and ask again." "Don't count on it."
  "My reply is no." "My sources say no."
  "Outlook not so good." "Very doubtful.")

echo "Oh! Magic 8 Ball, Please Tell Me True..." ; echo ""
/bin/echo -n "What is your question? "
read question

answer=$(( $RANDOM % 20 ))

echo ""
echo "I have looked into the future and I say: "
echo "     ${answers[$answer]}" ; echo ""

exit 0
```

Let's do a quick run to see if I'm the most popular *LJ* writer:

```
$ sh magic8.sh
Oh! Magic 8 Ball, Please Tell Me True...

What is your question? Am I the most popular LJ writer?

I have looked into the future and I say:
     My reply is no.
```

Ouch, that's harsh. I write the darn divination program, and it just drops a brick on my

foot. Yeesh.

More seriously, Magic 8 Ball is too simple to make an interesting shell script. By contrast, *Call of Duty* is way too complex, even if I did a version with text output instead of gorgeously rendered 3D graphics.

Card Game Function Library

That's why card games prove to be good as programming challenges or exercises: the core mechanism of a 52-card random deck is pretty straightforward, so it's all about the actual cardplay.

Not only that, but as I've written before about card games as shell scripts, I already have a handy set of functions to create, shuffle and display cards out of a deck. If you want to rummage in the archives, I've tackled *Acey-Deucey*, *Baccarat* and some bits and pieces of *Cribbage*.

In order to jump right into the new game that I'm going to describe how to build, *Go Fish!*, let's steal the following functions from my earlier scripts:

- `initializeDeck`
- `shuffleDeck`
- `pickCard`
- `showCard`

I've uploaded the script library to my AskDaveTaylor site, so you can grab it here if you like.

With those functions all available, let's start writing a *Go Fish!* game.

Heading to the Fishing Hole

If you don't have kids, it might have been a while since you've played the simple game of *Go Fish!*. I would categorize it as a memorization game, because if you can remember what the other player has asked, you'll generally win the game.

Go Fish! is pretty simple. Each player starts with seven cards, and they take turns asking the other player if they have one or more of a particular card rank. If they do, they hand over all of their cards of that rank. If they do not, then you "go fish" and take the top card from the deck.

Once you have four of a kind, you immediately place those in front of you. If you ever have no cards in your hand, you immediately pick two from the deck and play proceeds. The game ends when there are no cards left in either person's hand and none in the deck. The winner is the one with more sets at the end of the game. It's pretty simple.

Note: the game becomes a lot more interesting with three or four players. Try it!

To get started, here's an easy way to include a set of functions from another file:

```
cardlib="playing-card-library.sh"

if [ ! -f $cardlib ] ; then
  echo "Can't find the playing card library $cardlib"
  exit 1
fi

. $cardlib
```

The conditional test before the "." source statement might be redundant, but why not have a nice error message in the case that the cardlib file isn't found?

The source statement (with slightly different syntax in different shells) is interesting. It reads the specified file as if its contents were part of the current file. This means any functions, any variables, anything that's in the file is incorporated into the current shell, not a subshell (as would happen if you used `sh $cardlib`, for example).

Now that the functions are defined, it's time to deal seven cards to both the computer

player and the user. To track all this, I'm going to use two arrays: `myhand` and `yourhand`. The shell dynamically resizes arrays as needed, which is great for *Go Fish!*, because you could end up having a lot more than seven cards during the game:

```
i=1

while [ $i -lt 8 ] ; do
  myhand[$i]=${newdeck[$i]}
  yourhand[$i]=${newdeck[$(( $i + 7 ))]}
  i=$(( $i + 1 ))
done
```

There are a lot of punctuation symbols in this line:

```
yourhand[$i]=${newdeck[$(( $i + 7 ))]}
```

Unwrap it step by step, and you'll remember that any array reference must be `${name[x]}`, which is half the complexity. The other half is the `$(( equation ))` notation to do some basic math without a subshell. Basically, this is like the worst dealer in the world, dealing cards 1–7 to the computer and 8–14 to the player. But if it's shuffled...

The `showCard` function (part of `cardlib`) is a bit clunky, because it doesn't actually output the card name, it just preloads global variable `cardname` with the correct value. So here's how to show a hand:

```
echo computer hand:
showCard ${myhand[1]} ; echo "  $cardname"
showCard ${myhand[2]} ; echo "  $cardname"
showCard ${myhand[3]} ; echo "  $cardname"
showCard ${myhand[4]} ; echo "  $cardname"
showCard ${myhand[5]} ; echo "  $cardname"
showCard ${myhand[6]} ; echo "  $cardname"
```

```
showCard ${myhand[7]} ; echo "  $cardname"
```

Running this, it's clear that I might need to sort the cards by rank to make the game easier to play:

```
$ sh gofish.sh
computer hand:
  5 of Diamonds
  9 of Hearts
  4 of Clubs
  7 of Diamonds
  8 of Hearts
  K of Hearts
  K of Clubs
your hand:
  5 of Hearts
  9 of Diamonds
  Q of Diamonds
  2 of Spades
  6 of Clubs
  2 of Diamonds
  Q of Spades
```

Since the player won't ever have to specify an individual card, I think I can get away without sorting the hand, so I'll leave that as a task for you, dear reader, when you start fiddling with the code.

Won't need to specify a card? Right. You'll ask the computer if it has any, say, "twos" by entering a "2" at the prompt. In fact, to make this interesting, let's let the player potentially cheat by having the computer ask if they have a specific card, rather than automatically just taking it out of the hand.

I'm imagining a play sequence like this:

```
Your hand:
  5 of Hearts
  9 of Diamonds
  Q of Diamonds
  2 of Spades
  6 of Clubs
  2 of Diamonds
  Q of Spades

You go first. You ask me if I have: 3
** You don't have any 3s so you can't ask for that!
You ask me if I have: 2
** I do not. Go fish.
(( you pick up the 9 of Hearts ))

My turn. Do you have any 7s? (yes/no):
```

With that in mind, let's stop here and pick up this coding project next time. See you then, and in the meantime, find a youngling and play a few games of *Go Fish!* to see the (admittedly fairly minimal) nuances of the game. ■

Send comments or feedback via https://www.linuxjournal.com/contact or email ljeditor@linuxjournal.com.

What's New in Kernel Development

By Zack Brown



Zack Brown is a tech journalist at *Linux Journal* and *Linux Magazine*, and is a former author of the "Kernel Traffic" weekly newsletter and the "Learn Plover" stenographic typing tutorials. He first installed Slackware Linux in 1993 on his 386 with 8 megs of RAM and had his mind permanently blown by the Open Source community. He is the inventor of the *Crumble* pure strategy board game, which you can make yourself with a few pieces of cardboard. He also enjoys writing fiction, attempting animation, reforming Labanotation, designing and sewing his own clothes, learning French and spending time with friends'n'family.

Simplifying Function Tracing for the Modern GCC

Steven Rostedt wanted to do a little housekeeping, specifically with the function tracing code used in debugging the kernel. Up until then, the kernel could enable function tracing using either **GCC**'s `-pg` flag or a combination of `-pg` and `-mfentry`. In each case, GCC would create a special routine that would execute at the start of each function, so the kernel could track calls to all functions. With just `-pg`, GCC would create a call to `mcount()` in all C functions, although with `-pg` coupled with `-mfentry`, it would create a call to `fentry()`.

Steven pointed out that using `-mfentry` was generally regarded as superior, so much so that the kernel build system always would choose it over the `mcount()` alternative by testing GCC at compile time to see if it actually supported that command-line argument.

This is all very normal. Since any user might have any version of a given piece of software in the toolchain, or a variety of different CPUs and so on, each with different capabilities, the

kernel build system runs many tests to identify the best available features that the kernel will be able to rely on.

But in this case, Steven noticed that for **Linux version 4.19**, **Linus Torvalds** had agreed to bump the minimum supported GCC version to 4.6. Coincidentally, as Steven now pointed out, GCC version 4.6 was the first to support the `-mfentry` argument. And, this was his point—all supported versions of GCC now supported the better function tracing option, and so there was no need for the kernel build system to cling to the `mcount()` implementation at all.

Steven posted a patch to rip it out by the roots.

Peter Zijlstra gave his support for this plan, as did **Jiri Kosina**. And, Jiri in particular spat upon the face of the `mcount()` solution.

Linus also liked Steven's patch, and he pointed out that with **mcount()** out of the picture, there were several more areas in the kernel that had existed simply to help choose between `mcount()` and `fentry()`, and that those now also could be removed. But Steven replied that, although yes this should be done, he still wanted to do split it up into a separate patch, for cleanliness' sake.

As it turned out, Steven's patch actually applied only to the **x86** kernel port. A lot of other architectures still used `mcount()`, as **Josh Poimboeuf** pointed out. And Steven confirmed, "fentry works nicely when you have a single instruction that pushes the return address on the stack and then jumps to another location. It's much trickier to implement with link registers. There's a few different implementations for other archs, but mcount happens to be the one supported by most."

And that was that. Steven's patch certainly will go into the kernel as soon as it's fully ready. It's enjoyable to watch these details shake out, after the relatively large decision to change the minimum supported GCC version. I imagine there are several more areas of the kernel that can be simplified and cleaned up, now that they don't have to support older versions of GCC.

Shrinking Linux Attack Surfaces

Often, a kernel developer will try to reduce the size of an attack surface against Linux, even if it can't be closed entirely. It's generally a toss-up whether such a patch makes it into the kernel. Linus Torvalds always prefers security patches that really close a hole, rather than just give attackers a slightly harder time of it.

Matthew Garrett recognized that userspace applications might have secret data that might be sitting in RAM at any given time, and that those applications might want to wipe that data clean so no one could look at it.

There were various ways to do this already in the kernel, as Matthew pointed out. An application could use `mlock()` to prevent its memory contents from being pushed into swap, where it might be read more easily by attackers. An application also could use `atexit()` to cause its memory to be thoroughly overwritten when the application exited, thus leaving no secret data in the general pool of available RAM.

The problem, Matthew pointed out, came if an attacker was able to reboot the system at a critical moment—say, before the user's data could be safely overwritten. If attackers then booted into a different OS, they might be able to examine the data still stored in RAM, left over from the previously running Linux system.

As Matthew also noted, the existing way to prevent even that was to tell the **UEFI** firmware to wipe system memory before booting to another OS, but this would dramatically increase the amount of time it took to reboot. And if the good guys had won out over the attackers, forcing them to wait a long time for a reboot could be considered a denial of service attack—or at least downright annoying.

Ideally, Matthew said, if the attackers were only able to induce a clean shutdown—not simply a cold boot—then there needed to be a way to tell Linux to scrub all data out of RAM, so there would be no further need for UEFI to handle it, and thus no need for a very long delay during reboot.

Matthew explained the reasoning behind his patch. He said:

> Unfortunately, if an application exits uncleanly, its secrets may still be present in RAM. This can't be easily fixed in userland (eg, if the OOM killer decides to kill a process holding secrets, we're not going to be able to avoid that), so this patch adds a new flag to madvise() to allow userland to request that the kernel clear the covered pages whenever the page reference count hits zero. Since vm_flags is already full on 32-bit, it will only work on 64-bit systems.

Matthew Wilcox liked this plan and offered some technical suggestions for Matthew G's patch, and Matthew G posted an updated version in response.

Michal Hocko also had some technical suggestions, including the idea that the patch should not just wipe RAM, but also any swap space, for added protection.

But, **Christopher Lameter** replied to Matthew G's patch, saying that it didn't actually fix the problem, even if it made the attack more difficult to carry out. As he put it:

> The pages are cleared anyways when reallocated to another process. This just clears it sooner before reuse. So it will reduce the time that a page contains the secret sauce in case the program is aborted and cannot run its exit handling.
>
> Is that really worth extending system calls and adding kernel handling for this? Maybe the answer is yes given our current concern about anything related to "security".

Matthew G pointed out that if the system was mostly idle, no other process might claim the RAM that still held secret data. In this case, those secrets would sit unguarded. And if someone did reboot the system at that time, the secret data would be exposed.

A bunch of people contributed technical suggestions, and Matthew G submitted several new versions of his patch, before the discussion ended.

There's clearly some interest in this patch, but no one was singing about it on their way to the Grey Havens. It clearly represents a security improvement, in the sense that it makes the time window a bit tighter for an attacker to take advantage of

exposed data, but at the same time, that window does remain open for a certain amount of time. Hostile attackers could potentially take advantage of that to gain access to privileged data, even with Matthew G's patch. It's unclear to me whether or not this patch will go into the kernel.

Address Space Isolation and the Linux Kernel

Mike Rapoport from **IBM** launched a bid to implement address space isolation in the Linux kernel. Address space isolation emanates from the idea of virtual memory—where the system maps all its hardware devices' memory addresses into a clean virtual space so that they all appear to be one smooth range of available RAM. A system that implements virtual memory also can create isolated address spaces that are available only to part of the system or to certain processes.

The idea, as Mike expressed it, is that if hostile users find themselves in an isolated address space, even if they find bugs in the kernel that might be exploited to gain control of the system, the system they would gain control over would be just that tiny area of RAM to which they had access. So they might be able to mess up their own local user, but not any other users on the system, nor would they be able to gain access to root level infrastructure.

In fact, Mike posted patches to implement an element of this idea, called **System Call Isolation** (SCI). This would cause system calls to each run in their own isolated address space. So if, somehow, an attacker were able to modify the return values stored in the stack, there would be no useful location to which to return.

His approach was relatively straightforward. The kernel already maintains a "symbol table" with the addresses of all its functions. Mike's patches would make sure that any return addresses that popped off the stack corresponded to entries in the symbol table. And since "attacks are all about jumping to gadget code which is effectively in the middle of real functions, the jumps they induce are to code that doesn't have an external symbol, so it should mostly detect when they happen."

The problem, he acknowledged, was that implementing this would have a speed hit.

He saw no way to perform and enforce these checks without slowing down the kernel. For that reason, Mike said, "it should only be activated for processes or containers we know should be untrusted."

There was not much enthusiasm for this patch. As Jiri Kosina pointed out, Mike's code was incompatible with other security projects like retpolines, which tries to prevent certain types of data leaks falling into an attacker's hands.

There was no real discussion and no interest was expressed in the patch. The combination of the speed hit, the conflict with existing security projects, and the fact that it tried to secure against only hypothetical security holes and not actual flaws in the system, probably combined to make this patch set less interesting to kernel developers.

It's one of the less pleasant aspects of kernel development. Someone can put a lot of hours into a project, with no way to know in advance what objections might be raised at the end. It wouldn't have been obvious to Mike and his colleagues that a speed hit would be necessary. And the possibility of conflict with other existing kernel projects is always very difficult to predict, especially since there often are workarounds that can be discovered only once members of the two projects start debating the various issues in public.

Only Linus Torvalds' general reluctance to add security features that do not address existing security holes could have been predicted. He seems very consistent on that point, much to the annoyance of security-minded developers throughout the Open Source world. The idea of reducing the size of an attack surface seems self-evident to them; while to Linus, it seems self-evident that you shouldn't fix what isn't broken, especially where the fix adds bloat and increases the maintenance costs for the whole project. I think it's likely that even if Jiri and other developers had approved of Mike's patches, Linus might have objected later on.

Note: if you're mentioned in this article and want to send a response, please send a message with your response text to ljeditor@linuxjournal.com*, and we'll run it in the next Letters section and post it on the website as an addendum to the original article.* ■

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

DEEP DIVE

THE



Image by Jesse Ratterree

A Guide to Basic Command-Line Tasks

A whirlwind tour of the "the user interface that wouldn't die".

By Dave Taylor

Opening scene: a dusty 1950s-era computer room with spinning mag tapes and hundreds of flashing lights. There's an imposing grey teletype machine and empty chair front and center. To the right is a huge, messy pile of printouts clearly torn off the machine and marked up with pencils. It's late at night with minimal room lighting.

Cut to opening animation: "The User Interface That Wouldn't Die!"

Sf/x a scream as the screen bursts into a pure white...

Okay, so maybe using the command line isn't quite this dramatic as we move slowly but inexorably into the year 2020. Twenty twenty. Hard to believe we're already that far into the 21st century. And remember, the very first command lines were from the Multics era, if not earlier (Multics begat Unix which begat Linux many years later). We're talking about a user interface that's been around since the early 1960s.

This leads to the obvious question, "What's the darn appeal for people to use such an ancient interface when there are fancy graphical window managers, mice, touchscreens and swanky million-color displays?"

I occasionally ask myself this very question when I crack open a terminal window and

start typing at the command line, just to realize that the command-line interface is still popular *because it's so incredibly efficient*.

Want to list only files that have a "z" in their name? Want to check the last time that user "maria" logged in? Need to change all the filenames in that archive from your supplier so that everything's in lowercase? Want to bulk-resize thousands of images? Create an encrypted backup and automatically copy it to a cloud server? Log in to a customer's system 7,500 miles away for remote diagnostics?

Those are just a few of the millions of tasks you can accomplish with the command line in Linux, and it's exactly why smart users still are finding that CLI and using it. My guess is that a significant subset of *Linux Journal* readers type in a command at least weekly, if not much more frequently.

But, how well do you actually know the command line, the *user interface that wouldn't die*? Let's have a refresher of the commands folks likely use the most often since this is the command-line issue of the magazine.

Note: I've been writing about the command line for eons. Books I've written on the subject include Teach Yourself Unix in a Week (*which turned into* Teach Yourself Unix in 24 Hours *when a week seemed like too much time*), Learning Unix for MacOS X *and the ever-popular* Wicked Cool Shell Scripts.

Editing Commands and Command History

Before you start typing in commands, it's really helpful to know how you can tweak and modify your command line. Start with `history`, and you'll see a list of your previous commands. Each is prefaced by a number, and you can repeat an earlier command quickly and easily with the shortcut `!number`, as shown:

```
$ history
    1  PS1="$ "
    2  uptime
    3  history 10
```

```
    4  ls -F
    5  find . -name "*.c" -print
    6  who
    7  date
    8  clear
    9  history
$ !2
uptime
 08:16:03 up  1:11,  1 user,  load average: 0.17, 0.18, 0.18
$
```

Notice that when I did `!2` to repeat command #2, the shell then showed the command I'd matched (`uptime`) followed by the result of that command being invoked.

You also can repeat a command by using that same `!` followed by a letter or two, so I easily could repeat the `find` command (command #5) with this:

```
$ !f
find . -name "*.c" -print
...
```

That's the majority of my command-line manipulation, actually. On many systems, you also can use the cursor up and cursor down arrows to scroll through your command history, but that almost feels like cheating if I'm talking about a throwback to the teletype machine.

Trivia: /dev/tty is named after the teletype system, and it's still the mnemonic for your terminal session all these decades later. That teletype hasn't completely vanished!

There's quite a bit more you can do with the command line, including pulling arguments from one command into another and so on, but those things are a bit more obscure and probably not as helpful as just knowing that you always can check and access your history with just a keystroke or two. In fact, when developing a script,

I often find myself in a cycle of `!v` to edit and `!.` to execute it once I've typed in the full commands a single time.

Navigating the Filesystem

The most rudimentary use of the command line is to explore the filesystem. You surely already know that the Linux filesystem is an inverted hierarchical tree, right? *Yes, and don't call me Shirley!*

As a result, the key commands to learn for filesystem navigation are to identify your current location in the filesystem (`pwd`) and how to move around (`cd`):

```
$ pwd
/home/taylor
$ cd /home
$ pwd
/home
```

You can see that I started out with a *present working directory* of /home/taylor, then used the *change directory* command to move to /home, at which point my pwd is now, logically enough, /home.

There are two additional shortcuts for navigating the filesystem in Linux, and those are `..` to back up one level in the system and `~` as a shortcut for your home directory. Your home directory, just to be clear, is where you start when you've logged in to the system.

This should all make sense:

```
$ cd ~
$ pwd
/home/taylor
$ cd ..
$ pwd
/home
```

That's the basics of moving around. Pretty darn easy, really.

Listing Files and Directories

Moving around in the filesystem is of limited value if you can't actually look at what's in your current location. That's what the `ls` command is for, and if there's one command that you should grow to love, it's `ls`. This is also the first command I'm considering here that has arguments and command flags—*lots* of them actually. But to start, at its most basic:

```
$ cd ~
$ ls
Desktop    Downloads         Music     Public     Videos
Documents  examples.desktop  Pictures  Templates
```

You can see that there are eight entries in my home directory (remember, `~` is a shortcut for home), but what are they? My notational convention is to have directories start with an uppercase letter and have files all be lowercase, but that doesn't mean I'm 100% consistent.

To find out, add the `-F` flag to the `ls` command, which appends a symbol suffix to indicate file or directory type:

```
$ ls -F
Desktop/    Downloads/        Music/     Public/     Videos/
Documents/  examples.desktop  Pictures/  Templates/
```

Everything with a trailing / is a directory.

This is a good moment to point out something pretty important. Although we see directories and files as being quite different, the Linux filesystem views them all as just objects or entities. You can rename a directory the same way you can rename a file, for example, and directories and files can move around in just about identical fashions too. In fact, Linux has a sloppy habit of using the filename suffix to identify the type of

file too, so rename "resume.docx" to "resume.mp3", and suddenly it'll try to play your résumé instead of letting you edit it.

The `-F` flag is useful, but `-l` (that's a lowercase L) is much more useful as it offers up what's known as a "long listing" format:

```
$ ls -l
total 44
drwxr-xr-x 2 taylor taylor 4096 Mar  9 06:48 Desktop
drwxr-xr-x 2 taylor taylor 4096 Mar  9 06:48 Documents
drwxr-xr-x 2 taylor taylor 4096 Mar  9 06:48 Downloads
-rw-r--r-- 1 taylor taylor 8980 Mar  9 06:44 examples.desktop
drwxr-xr-x 2 taylor taylor 4096 Mar  9 06:48 Music
drwxr-xr-x 2 taylor taylor 4096 Mar  9 06:48 Pictures
drwxr-xr-x 2 taylor taylor 4096 Mar  9 06:48 Public
drwxr-xr-x 2 taylor taylor 4096 Mar  9 06:48 Templates
drwxr-xr-x 2 taylor taylor 4096 Mar  9 06:48 Videos
```

Now you can see permissions, ownership, group ownership, size (sort of), last modified dates and the file or directory name. Confusingly, the meaning of these values is different based on the type of file or object represented. All directories are shown with basically the same size (4K here), which is basically useless, but notice that the lone file, examples.desktop, has a different value. That's its actual size: 8,980 bytes.

It's the permissions string that's more interesting, however. Linux permissions model UNIX permissions and are based on three concentric circles of access: owner, group and everyone. To decode one of the permissions strings, know that the first character indicates type, then each three-character set is read/write/execute permission status for each of the three circles of access.

The directory Desktop, for example, has a d as its first letter. Then the *owner* of the file (taylor, as shown in column 4) has read+write+execute permission as denoted by `rwx`. The *group* to which the directory is assigned (also taylor in this example,

as shown in column 5) has read+execute permission, as denoted by `r-x`. Finally, everyone else on the system has read+execute only permission, meaning that only the owner of the directory can edit or change it.

Similarly, the file examples.desktop has read+write permission for its owner, taylor, and everyone else has read-only permission and cannot edit or change it or its content. That's what -rw-r--r-- means. Focus on the three-letter sequence * three access levels, and it shouldn't be too intimidating. But there's no way around it, permissions for directories are a bit more confusing. Suffice it to say, if you want someone to have access, make it `r-x`, and if you want them to be able to change, add and edit, use `rwx`. Close it off? That's what `--` is for.

Let's check out the topmost directory for some more interesting examples of permissions:

```
$ cd /
$ ls -l
total 970072
drwxr-xr-x   2 root root      4096 Mar  9 06:52 bin
drwxr-xr-x  19 root root      4060 Mar  9 06:56 dev
drwxr-xr-x 124 root root     12288 Mar  9 07:03 etc
drwxr-xr-x   3 root root      4096 Mar  9 06:44 home
lrwxrwxrwx   1 root root        33 Mar  9 06:54 initrd.img ->
boot/initrd.img-4.18.0-16-generic
drwxr-xr-x  21 root root      4096 Mar  9 06:46 lib
drwxr-xr-x   2 root root      4096 Oct 17 16:23 lib64
drwx------   2 root root     16384 Mar  9 06:42 lost+found
dr-xr-xr-x 258 root root         0 Mar  9 06:56 proc
drwx------   3 root root      4096 Oct 17 16:34 root
drwxr-xr-x  31 root root       880 Apr  8 08:05 run
drwxr-xr-x   2 root root      4096 Oct 17 16:23 sry
-rw-------   1 root root 993244160 Mar  9 06:42 swapfile
```

I've pruned the above just a bit to make it less overwhelming, but notice how much

variation there is! The first letter of the permissions string can also be an "l" (lowercase L) to indicate a symbolic link. In this instance, it shows that the entry /initrd.img is actually a link to /boot/initd.img-4.18.0-16-generic. Symbolic links are super small too, as they need to contain only the destination filename. This one's 33 bytes in size.

Notice the permission for lost+found: `rwx` for owner (root) and completely shut off for everyone else. Oh, and see that swapfile? How big is it?

Fortunately, there's one more `ls` flag worth mentioning: `-h`. That offers a "human-friendly" size. Add that to the ability to specify a directory or even a single file on the command line, and you can figure it out quickly:

```
$ ls -lh /swapfile
-rw------- 1 root root 948M Mar  9 06:42 /swapfile
```

Between `cd` and `ls`, you now can move around the filesystem quickly.

Tip: when you're typing in a file or directory name, try pressing the Tab key to expand it. As long as it will expand to a unique word or name, that's all you need to do, so typing `/sw` *and then pressing Tab would work fine in the above example. Handy indeed!*

Moving and Copying Files and Directories

Next up in this quick tour of the command line are the commands that let you move and copy files and folders, and the command that lets you create new directories. Moving and copying at some level are the same task, a stream of bytes moving into a new file container, but the big difference is what happens to the original file. With a move, the source file is deleted.

Let's jump back to my home directory and move that examples.desktop file to the Desktop directory:

```
$ cd ~
$ ls -l examples.desktop
```

```
-rw-r--r-- 1 taylor taylor 8980 Mar  9 06:44 examples.desktop
$ mv examples.desktop Desktop/
$ ls -F
Desktop/  Documents/  Downloads/  Music/  Pictures/  Public/
Templates/  Videos/
```

There's another handy shortcut demonstrated here. If you move or copy a file into a new directory, you can just specify the destination directory, and it'll retain its name. This means that, as shown, there's no longer a file called examples.desktop in my home directory.

You can rename it as you move or copy a file or folder, of course, so this would work fine:

```
$ mv Videos Desktop/my-video-archive
$
```

And this would work too:

```
$ cp Desktop/examples.desktop my-example
$
```

In fact, if you want to rename a file, it's the `mv` command you'll use, even if you're "moving" it within the same directory:

```
$ mv my-example demo-example.txt
$
```

There's no rename command in Linux, nor do you need one.

Of Wildcards, Asterisks and Quotes

In the old days, UNIX commands were all lowercase and filenames also were all lowercase and never had spaces within them. So it was easy to `mv test test.c`

and know it'd work properly. Modern graphical interfaces work fine with more complex names, so it's no surprise that there now are multiword directories and filenames.

To deal with complex names, quote things, ideally with the double quotes:

```
$ ls
examples.desktop
$ mv examples.desktop "My Favorite Examples"
$ ls -l My\ Favorite\ Examples
-rw-r--r-- 1 taylor taylor 8980 Mar  9 06:44 'My Favorite
 ↪Examples'
```

Specific characters that cause trouble can be escaped with a prefacing backslash, as the second `ls` command shows. Better yet, that's from me doing a Tab expansion, so the shell's smart enough to do this all by itself. Helpful!

The shell also has a cool wildcard feature, which is great if you want to do any sort of bulk moving. For example, C program files are denoted with a ".c" suffix, so a directory full of C source files might look like this:

```
$ ls -F
demoprog*  libalt.c  library.c  main1.c  main2.c  main.c
Makefile  README  utilities.c
```

Notice in this instance that the `-F` flag has given a "*" suffix to indicate that `demoprog` is executable.

To reference all the C source files en masse, you could use `*.c` as the notation. Copy them all into a backup directory? Easy:

```
$ cp *.c ../backupdir
```

This proves generally true, so doing this:

```
$ cp * ../backupdir
```

will copy *everything from this directory*, including the executable. There's lots more to the wildcard language of the shell, including `?` to match a single letter (so main?.c will match main1.c and main2.c, but not main.c) and sets like `[MR]*` to match M* and R*, but the asterisk wildcard will definitely get you started.

Pipes and Redirects

Think about this: every command invoked on the command line automatically has an input, an output and an error device associated. By default, the input is the keyboard, and the output and error output are both the screen or terminal. But, you can change them.

To demonstrate, I'm going to introduce another command: `wc`. The `wc` command does not point you to the closest restroom when you're in the UK (yeah, bad joke!), it offers up the *word count* for a file. For example:

```
$ wc main.c
  44  129 1172 main.c
```

This shows you that there are 44 lines, 129 words and 1172 characters in the C source file main.c—useful. Add `-w`, and it'll just report words too: `wc -w main.c`.

But, this command also can use the < symbol to redirect input. Notice how it subtly changes the results:

```
$ wc -w < main.c
129
```

Because I used redirection, it didn't know the name of the input file. You can use `<` to redirect input, but you also can use `>` to redirect the output of the command and even `>>` to redirect output and append it to an existing file. Consider this:

```
$ date > uptime.log
$ uptime >> uptime.log
$
$ cat uptime.log
Mon Apr  8 12:17:49 MDT 2019
 12:17:53 up 14 min,  1 user,  load average: 0.23, 0.41, 0.45
$
```

Oops, the `cat` program—short for concatenate—dumps the contents of a file—any file, even one that's not actually readable. In the above, the date command created the output file (or overwrote it if it already existed, which is a danger if you mess up input and output files), then the second command had its output appended. Finally, `cat` showed what's in uptime.log.

If you can change the input and output of a command, can you have one command's output be the input to another command's input? That's what *pipes* are for, and it's one of the most powerful—and wicked cool—capabilities of the command line. There are hundreds of commands with thousands of flags and options, but once you realize that one command's output can be another's input, and that output can be hooked to yet another, well, then you realize there are *millions* of different commands you can form with a few dozen keystrokes.

At its most simple, how about this:

```
$ uptime | wc
      1      10      61
$ uptime | wc | wc
      1       3      24
```

See what I did in that second command? I used `wc` the second time to count the number of words, lines and characters that were the output of the first `wc` invocation. It's kind of a weird example, but it highlights that any command that's in a pipe acts independently, so the second `wc` couldn't know that it was being invoked twice.

The combination of pipes and redirection offers an extraordinarily rich environment within which you can create just about any possible command you can imagine. That's where a lot of the fun of working with the Linux command line comes from. The other thing that makes it great fun for me is...

Shell Script Programming

Yep, that should be no surprise since I have been writing the Shell Script Programming column here in *Linux Journal* for more years than I want to remember. Imagine that everything you can do on the command line you can easily do within a simple script. Add conditional statements, functions and variables—the sky's the limit on what you can do with shell scripts, and I've written hundreds of different scripts, ranging from a few lines to hundreds of lines of complex code.

You can see 101 of my best and most interesting in my book *Wicked Cool Shell Scripts*, or just read my column here in the magazine. Or even better, do both!

I'm way out of space, which is too bad, because I could keep going for many, many more pages. Indeed, I have, which is why I've written an introduction to the Linux command line three times in different books. None were shorter than 300 pages either. Still, I hope this has whetted your appetite to learn more about the fantastic Linux command line, and remember that both MacOS X and Windows have command-line interfaces too.

It really is "the User Interface that Wouldn't Die" after all. ■



Dave Taylor has been hacking shell scripts on UNIX and Linux systems for a really long time. He's the author of *Learning Unix for Mac OS X* and *Wicked Cool Shell Scripts*. You can find him on Twitter as @DaveTaylor, and you can reach him through his tech Q&A site: Ask Dave Taylor.

Without a GUI—How to Live Entirely in a Terminal

Sure, it may be hard, but it is possible to give up graphical interfaces entirely—even in 2019.

By Bryan Lunduke

About three years back, I attempted to live entirely on the command line for 30 days—no graphical interface, no X Server, just a big-old terminal and me, for a month.

I lasted all of ten days.

Why did I attempt this? What on Earth would compel a man to give up all the trappings and features of modern graphical desktops and, instead, artificially restrict himself to using nothing but text-based, command-line software, as if he were stuck in the early 1980s?

Who knows. Clearly, I make questionable decisions.

But you know, if I'm being honest, the experience was not entirely unpleasant. Sure, I missed certain niceties from the graphical side of things, but there were some distinct benefits to living in a shell. My computers, even the low-powered ones, felt faster (command-line software tends to be a whole lot lighter and leaner than those with a graphical user interface). Plus, I was able to focus and get more work done without all the distractions of a graphical desktop, which wasn't bad.

What follows are the applications I found myself relying upon the most during those fateful ten days, separated into categories. In some cases, these are applications I currently use over (or in addition to) their graphical equivalents.

Quite honestly, it is entirely possible to live completely without a GUI (more or less)—even today, in 2019. And, these applications make it possible—challenging, but possible.

Web Browsing

Plenty of command-line web browsers exist. The classic Lynx typically comes to mind, as does ELinks. Both are capable of browsing basic HTML websites just fine. In fact, the experience of doing so is rather enjoyable. Sure, most websites don't load properly in the "everything is a dynamically loading, JavaScript thingamadoodle" future we live in, but the ones that do load, load *fast*, and free of distractions, which makes reading them downright enjoyable.

But for me, personally, I recommend w3m.

w3m supports inline images (via installing the `w3m-img` package)—seriously, a web browser with image support, inside the terminal. The future is now.

It also makes filling out web forms easy—well, maybe not easy, but at least doable—by opening a configured text editor (such as nano or vim) for entering form text. It feels a little weird the first time you do it, but it's surprisingly intuitive.

Email

Email is another one of those things you simply can't live without. Luckily, people were emailing each other from UNIX/Linux machines long before modern graphical email clients (or webmail) even existed.

There are a few quality options for sending mail from the comforts of your terminal (including Mutt and Notmuch), but for my money, it doesn't get much better than Alpine. It is, by far, the most approachable and learnable. Common hot-keys are


Your continued donations keep Wikipedia running!

Main Page

From Wikipedia, the free encyclopedia

Jump to: navigation, search

Welcome to Wikipedia, the free encyclopedia that anyone can edit.

In this English version, started in 2001, we are currently working on 1,024,885 articles.

Overview - Questions - Categories - A?Z - Portals - Site news - Donations
Arts | Biography | Geography | History | Mathematics | Science | Society | Technology

Today's featured article

The Palazzo Pitti is a vast, mainly Renaissance palace in Florence, Italy. It is situated on the south side of the River Arno, a short distance from the Ponte Vecchio. The core of the present palazzo dates from 1458 and was originally the town residence of Luca Pitti, an ambitious Florentine banker. It was later bought by the Medici family in 1549: as the official residence of the ruling families of the Grand Duchy of Tuscany, it was enlarged and enriched almost continually over the following three centuries. In the 19th century, the palazzo, by then a great treasure house, was used as a power base by Napoleon I, and later served for a brief

In the news

Lennart Meri

* Former President of Estonia Lennart Meri (pictured) dies.
* Venezuela adopts a new flag.
* Former Yugoslav leader Slobodan Milo?evi? dies in his prison cell in The Hague, Netherlands.
* NASA's Mars Reconnaissance Orbiter enters Martian orbit.
* The Laotian rock rat is identified as part of a group previously thought to have disappeared 11 million years

<=UpDn Viewing <Main Page - Wikipedia, the free encyclopedia>

Figure 1. Browsing Wikipedia with Inline Images Using w3m

displayed right on screen (such as press r to reply to a selected email), and the overall layout and interface is going to be immediately recognizable to anyone who has used a graphical email client or webmail.

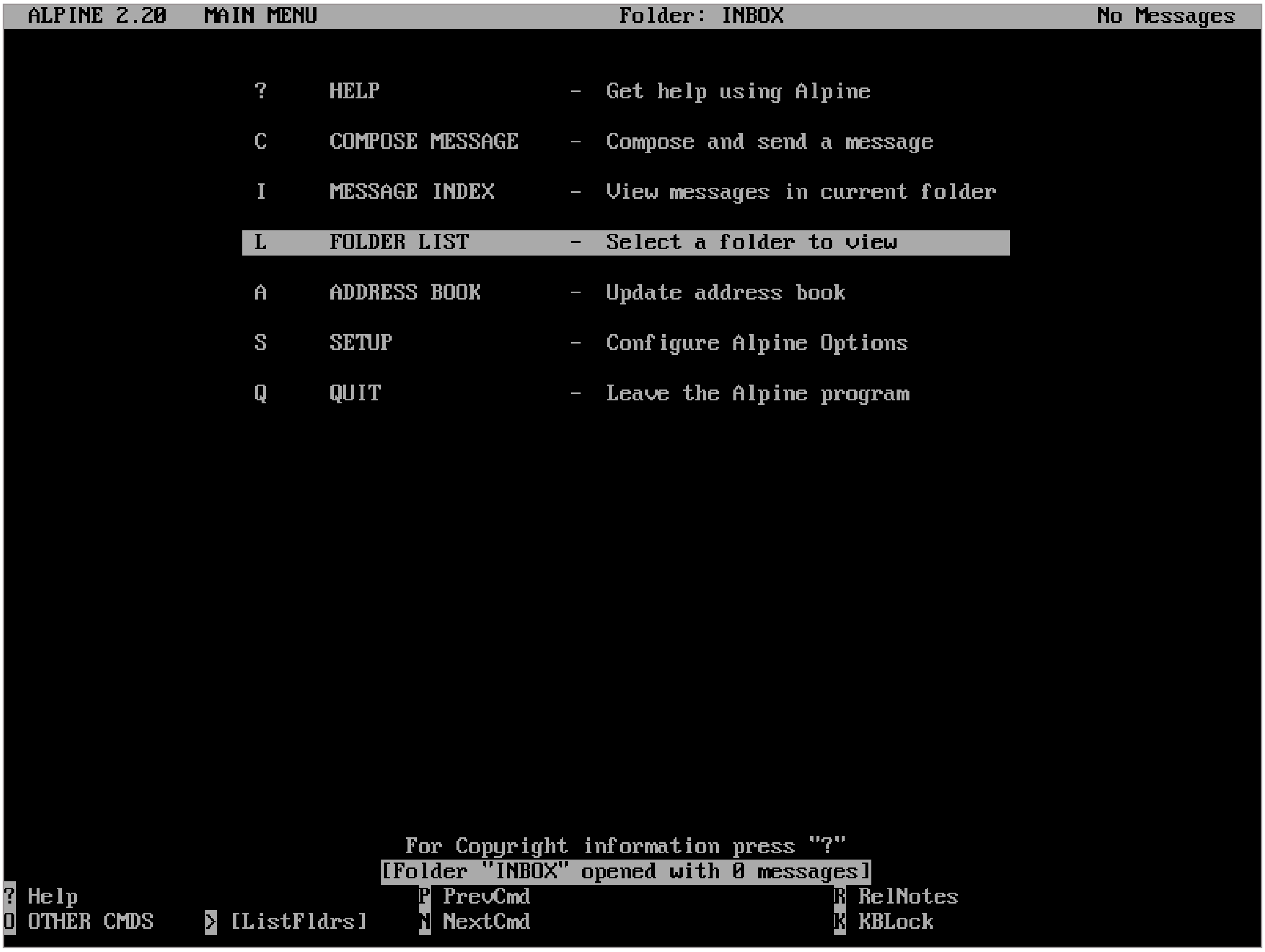


Figure 2. The Alpine Email Client Main Menu

Some other clients (like Mutt) also are quite excellent, but they have a bit steeper of a learning curve.

Instant Messaging

Wouldn't it be great if there was something like Pidgin, but entirely command-line based? Something that supports multiple chat protocols (like IRC, XMPP and so on) and lets you have chats with multiple people at once?

It turns out there is. It's called Finch, and it's made by the Pidgin folks.

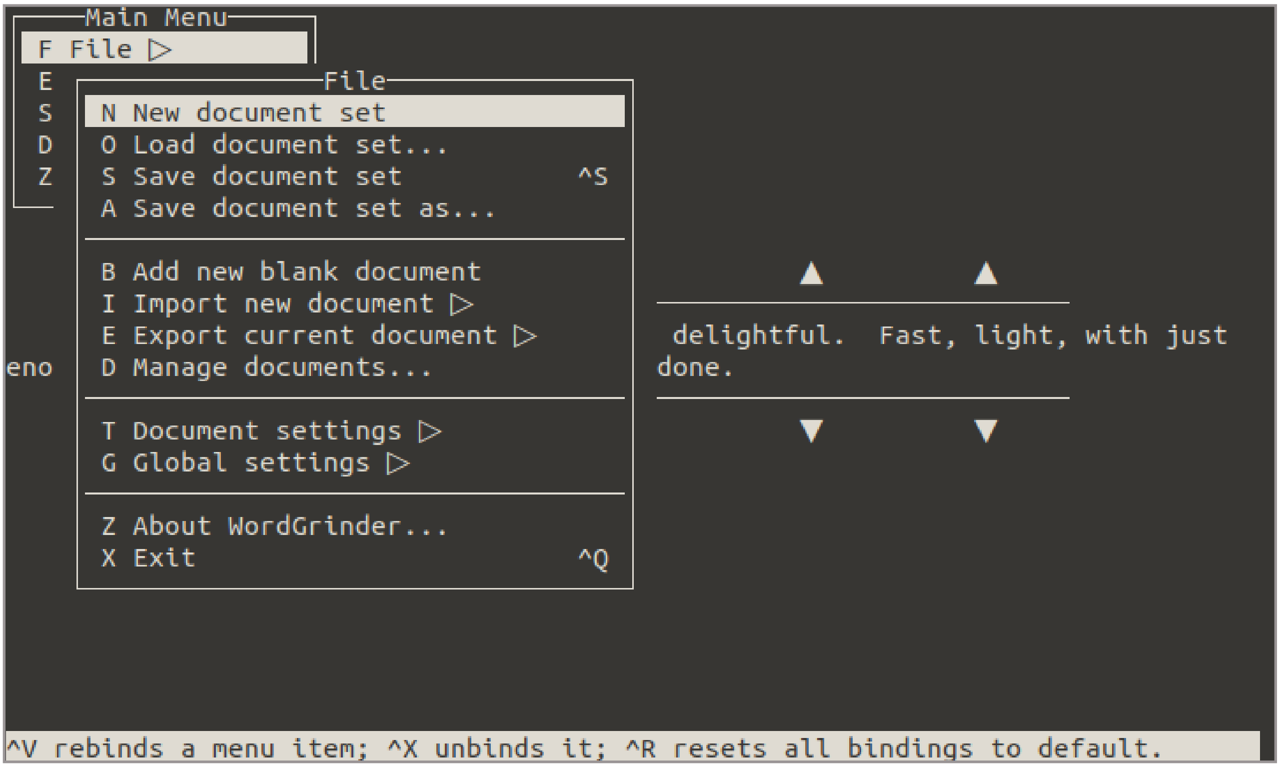


Figure 3. The File Menu in WordGrinder

It even has a nice TUI (Text User Interface) that somewhat mimics the GUI of Pidgin. Navigating around that interface without a mouse can be a bit confusing at first (Alt-N goes to the next "window", Alt-P to the previous and so forth), so taking a few minutes to learn the keyboard commands will save you much frustration.

It's not a perfect solution, mind you. Some of the protocols that Finch supports are either dead or dying (such as AOL Instant Messenger), but the support that is there works surprisingly well.

Word Processing

WordGrinder is the terminal-based word processor you've been looking for.

WordGrinder has a simple interface (press Escape to bring up a global menu with all the functions and feature available), is fast as heck, and is just downright

delightful to write in.

The darn thing even supports writing to ODT, HTML and Latex, and it can import ODT, HTML and text. There's even some basic formatting options (italic, bold, underline, margins), which is a bit surprising for a terminal-based word processor.

Spreadsheets

Text-based spreadsheet programs are nothing new. Heck, the fabled VisiCalc had an entirely text-based user interface, as did most spreadsheet programs built for a decade after that point.

Just the same, the spreadsheet options when running in Linux terminal are...limited.

The most commonly used option is known as sc (for Spreadsheet Calculator). It's been around for a good long time and works wonderfully well—for what it does. Enter text into a cell? Check. Enter numbers into a cell with some computation options? Double check.

Want to open a spreadsheet created in something like LibreOffice? Well, you're going to be plum out of luck there. sc uses a custom file format. Why not simply default to using CSV or some other standardized, text-based format for spreadsheets? I couldn't tell you. That confuses me too.

But luckily, the sc file format isn't too far off from CSV, and folks around the internet have created a variety of scripts to help convert between the two. It's not a seamless workflow, but sc is usable—once you learn the key bindings and get some supporting scripts in place to convert files.

Presentations

Yes! You can create and give presentations entirely from the terminal! It really works! I mean, there's not going to be any images, but who needs pictures in this day and age?

The program is called "tpp" (Text Presentation Program), and it's in just about

every repository on the planet. It has a bit steeper requirements than many terminal applications, as it relies on Ruby, but it's still lighter and faster than any graphical presentation program out there.

Presentations are created, in whatever text editor you choose (I'm a nano man, myself), and they're saved in a simple sort of markdown language. It has a very low learning curve.

So, maybe you won't be giving a presentation using tpp to a bunch of folks who expect copious clip art on your slides, but for a nice, nerdy audience? A purely text-based presentation will, if nothing else, win you a few high fives.

File Management

Copying files around in the terminal isn't exactly difficult once you learn a few commands (like `cp`, `mv` and `rm`), but having a nice interface to browse and copy files in bulk is simply a must.

And the very best option, in my humble opinion, is Midnight Commander, which is also known as simply mc. An open "clone" of the famous Norton Commander file browser, mc is one of the first applications I install on a new Linux system. It's simply the king of terminal file management.

Music

cmus is the music player you want to be using. It's light. It's fast. It's easy to use. It supports just about any audio format you could ever want (everything from MP3s and Oggs to MOD and SHN), plus streaming formats and playlists. It's just...the best.

Window Management

Just because you're living in a terminal doesn't mean you need to give up running (and looking at) multiple applications at the same time.

What one would call a window manager in a graphical desktop, in terminals is called a terminal multiplexer—same idea, more or less.

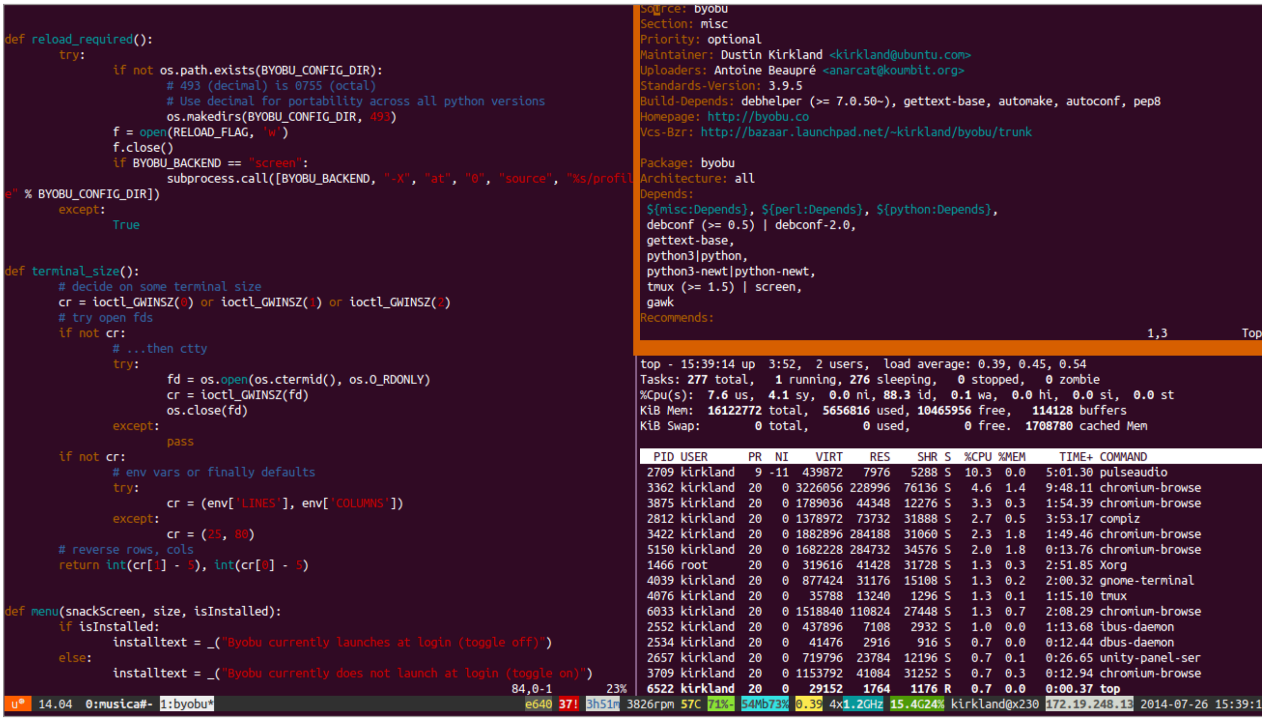


Figure 4. Three Terminal “Windows” Open in Byobu

There are three terminal multiplexers that most people tend to use: GNU Screen, tmux and Byobu.

GNU Screen and tmux are, for most people, fairly interchangeable. Once you figure out the keyboard shortcuts for creating new “windows” and moving between them, you’re all set.

Byobu (which actually utilizes Screen and tmux behind the scenes) aims to be a bit more full featured, emulating aspects of a traditional desktop environment and providing just a bit more visual information. Although it does tend to be slightly more finicky than the others.

Personally, I love tmux and use that one almost exclusively. Just do yourself a favor and really read through the man page first. Otherwise, you might end up pulling out a fair bit of hair as you learn how to switch between applications.

Really? No GUI? In 2019?

Seriously, it's doable. Sure, there are pain points—some pretty big ones at that. But, there's also something magical about it. Then again, maybe that's just the nostalgia centers of my brain talking.

Regardless, having these sorts of tools available (even if just to use when SSHing into a server) is downright handy. ■



Bryan Lunduke is a former Software Tester, former Programmer, former VP of Technology, former Linux Marketing Guy (tm), former openSUSE Board Member... and current Deputy Editor of *Linux Journal*, Marketing Director for Purism, as well as host of the popular *Lunduke Show*. More details: http://lunduke.com.

Resources

- Lynx
- ELinks
- w3m
- Mutt Email Client
- Notmuch—Just an email system
- Alpine
- Finch
- WordGrinder
- sc, Spreadsheet Calculator
- tpp, Text Presentation Program
- Midnight Commander
- cmus
- GNU Screen
- tmux
- Byobu

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

How to Expand Your Command-Line Scripting Options with Tcl

Get started scripting with Tcl, the Tool Command Language—this actually *is* your father's Oldsmobile.

By Mitch Frazier

The Tcl scripting language has been around a long time, and it tends to keep a pretty low profile. In fact, it's so long and so low, if you're not of a certain age and not a language junkie, you may never have even heard of it.

Despite its lack of headlines, Tcl still has a "vibrant community" (according to its **website**), and it continues to evolve, albeit slowly: versions 8.6.8 and 8.6.9 were released 11 months apart.

Tcl stands for Tool Command Language, and it originally was designed to be used as a language for embedding inside other applications. It still can be used for that, but it also has found success as a standalone scripting language.

If you've ever used or heard of git (who hasn't at least heard of it), the default GUI that comes with git is written in Tcl, using Tcl's GUI toolkit Tk.

If you do any embedded programming with ARM CPUs, the popular tool OpenOCD uses Tcl as its embedded programming language.

Tcl is not a scripting language like Bash in the sense that it does not "script" Linux commands.

In other words, when writing Tcl, you aren't generally executing `grep` and `sed` or any other commands that you normally would type at the command line, you're executing the commands that are built in to Tcl.

In this sense, Tcl is more akin to Python than to Bash.

In this article, I want to do five things:

1. Provide a "drink from the firehose" introduction to Tcl.

2. Take a quick look at some of the commands that are built in to Tcl.

3. Take an even quicker look at the Tk toolkit.

4. Show how to run Tcl scripts and introduce TclKits.

5. Explain how to use TclKit to create single file applications containing scripts and data files.

The Tcl Language

You can assign values to variables:

```
set a_num         99
set a_string      "some string"
set a_list        { 99 100 101 102 }
set an_array(0)   12
set an_array(1)   13
```

As you might expect for a scripting language, comments are lines that start with a hash sign (#). But contrary to what you might expect, hash signs on the same line as code are not seen as comments:

```
# This is a comment.
set a  12     # WRONG: this is not a comment
```

Within a "value", three types of substitution are done:

1. Variable substitution (for example, $name is replaced with the value of the variable name).

2. Command substitution (for example, [cname...] is replaced with the return value of the command cname). This is akin to Bash's backticks or its $(...) syntax.

3. Backslash substitution (for example, \<char> is replaced with the character <char>).

For example:

```
# Set a to 99 and b to 100
set a   99
set b   [expr $a + 1]
set c   \n
```

In set b above, the command [expr ...] evaluates its arguments as an expression and returns the result (all the "normal" operators are available).

All of these substitutions also work inside double-quoted strings:

```
set a  99
set b  "  a is $a\n  setting b to [expr $a + 1]"
```

Somewhat unexpectedly though, no substitutions are done in lists:

```
# WRONG: the first element of a_list will be "$a" not 99.
set a       99
set a_list   { $a 100 101 102 }
```

If you want to do the above, you need to use the `list` command:

```
# RIGHT: the first element of a_list will 99.
set a       99
set a_list   [list $a 100 101 102]
```

The reason for this strangeness will become clear in a bit.

Tcl also has control statements:

```
# if statements:
if { $a > 20 } {
    puts "a is greater than 20"
} elseif { $a > 10 } {
    puts "a is greater than 10"
} else {
    puts "a is less than or equal to 10"
}

# loop statements:
foreach var $list {
    puts "$var"
}
foreach var {1 2 3 4} {
    puts "$var"
}
```

Tcl has functions as well:

```
proc myfunc {arg1 arg2} {
    puts "arg1 is $arg1"
    puts "arg2 is $arg2"
}

# Call function:
myfunc 12 13

proc mysum {a b} {
    return [expr $a + $b]
}

# Capture return value of function:
set sum  [mysum 12 13]
```

Note that in the examples above, the curly brace that starts a “block” of code must be on the same line as the previous part of the statement:

```
########
# WRONG:
########
if { $a > 20 }
{
}
elseif { $a > 10 }
{
}
else
{
}
```

```
foreach var $list
{
}

proc myfunc {arg1 arg2}
{
}
```

To overcome this, you can can escape the newline before the start of the block:

```
###########################
# OK (but not very Tclish):
###########################
if { $a > 20 } \
{
}
elseif { $a > 10 } \
{
}
else \
{
}

foreach var $list \
{
}

proc myfunc {arg1 arg2} \
{
}
```

Again, the reason for this strangeness will be revealed shortly.

So except for using curly braces `{...}` around control statement conditions and formal function parameters, there's nothing too syntactically strange in Tcl.

Speaking of syntax, let's take a step back and look at Tcl's syntax in general.

It's only a slight exaggeration to say that Tcl's syntax can be defined in a single line:

```
WORD...
```

I know what you're thinking: "wait, you just showed me if statements, loops and functions, so how can one word and some ellipses define Tcl's syntax?"

Obviously, syntax alone doesn't define a language; you also need to understand the semantics or meaning of the syntax.

First, it's important to know what a `WORD` is in Tcl:

- A normal unquoted sequence of characters like the words in this sentence or most any sequence of characters without spaces.
- A string in double quotes.
- Text between matching opening and closing square brackets ([]).
- All the text, including newlines between match opening and closing curly braces ({}).

In all but the last item above, backslash substitution, variable substitution and command substitution is done. For the first item above, somewhat unexpectedly perhaps, this means unquoted words can include `$var` and `[command]` substitutions (as well as backslash substitutions); see the examples below.

Curly braces are a bit like Python's triple quotes (`""" ... """`).

Some sample words:

```
avar
some_name

# if b has the value 12, this will be the word "a12"
a$b

# if a is 1 and b is 2, this will be the word "c3"
c[expr $a + $b]

"double quoted string"
"$vars [commands] and backslash\n expansion is done"

{ this is one word }
{   this
    is
    one
    word
    too
}
```

I mentioned earlier that a `WORD` can be "most any sequence of characters without spaces", so you also can write code like the following, although you won't make many friends doing stuff like this:

```
set a'b'c       12
set a\"bcd\"e   99
```

The second thing to understand about the semantics of Tcl is that in Tcl everything is a command.

The first word on a line is the command, and the words that follow are the arguments

to the command. So, the truth is that Tcl doesn't have an "if statement"; it has an "if command" that looks like this:

```
if COND_WORD THEN_CODE_WORD optional-elseif-else-words
```

And since `WORD`s can be curly-brace blocks that span lines, `if` commands look like they should. So the following `if` command:

```
if { ... } {
    ...
} else {
    ...
}
```

Consists of five `WORD`s:

1. The literal `if`.

2. The `if` condition between braces `{ ... }`.

3. The "then" code block `{\n ... \n}`.

4. The literal `else`.

5. The `else` code block `{\n ... \n}`.

Okay, so there are no `if` statements, just `if` commands. At this point, you're probably thinking that this seems like a distinction without a difference, and most of the time it is, but not always.

For instance, let's say I'm working with a bunch of dates, and I have a bunch of code that checks to see if my date value is on a Monday:

```
# [clock scan ...]    - converts a string to a time
# [clock format ...]  - similar to Linux's "date +FORMAT"
if { [clock format [clock scan $date] -format %A] ==
 ↪"Monday" } {
    puts "It's Monday"
    # ,..
}
```

And further, let's say I get a bit tired of typing all that, so I decide to do this:

```
proc if_monday {date block} {
    set day [uplevel clock format \[ clock scan $date \]
     ↪-format "\{%A\}" ]

    if { $day == "Monday" } {
        uplevel $block
    }
}
```

Now I can just do the following:

```
if_monday { $date } {
    puts "It's Monday"
    # ,..
}
```

In other words, I just added a new command named `if_monday` that looks just like an `if` "statement" (for the sake of simplicity, I'm going to ignore the else part here).

The secret sauce here is the `uplevel` command. `uplevel` evaluates its arguments (the if expression or the if code block) in the context of the caller of the `if_monday` command. To see what this means, consider this implementation and its usage:

```
# WRONG:
proc if_monday {date block} {
    set day [uplevel clock format \[ clock scan $date \]
     ↪-format "\{%A\}" ]

    if { $day == "Monday" } {
        # WRONG
        eval $block
    }
}

set var 99
if_monday { $date } {
    puts "It's monday"
    set var 100
}

# var will have the value 99 not 100
puts "$var"
```

Since the `if_monday` command evaluated the code block using `eval` rather than `uplevel`, the `set var 100` that's inside the code block sets a variable named `var` in the `if_monday` procedure and not the `var` that's right before the call to `if_monday`.

On the other hand, if you use `uplevel` instead of `eval`, the correct version of the variable gets set—the one that's in the "context of the caller".

Not to beat a dead horse, but again, commands are just a sequence of words. Given these stub functions:

```
proc start  {}  { puts start }
proc stop   {}  { puts stop  }
```

I also can write my `if` statements like this:

```
if 1 start else stop
if [expr 1 == 0] start else stop
```

The `if` condition and the code blocks don't have to be inside curly braces; they just need to be something that Tcl considers to be a `WORD`.

Note, however, that there's a subtle, probably undesirable side effect of doing stuff like this. Consider the code:

```
proc test   {v} { puts $v; return 1 }

if 1 start elseif [test 44] { puts "elseif" } else stop
```

If you run this, the output will be:

```
44
start
```

Because the `elseif` condition is not inside curly braces, it is evaluated "before" the `if` command (so that the result can be passed as one of the arguments to the `if` command). So, although you can skip the curly braces, don't.

Earlier I mentioned there were a number of things that seemed strange and that I'd get back to them. Here they are:

1) Putting the code block or key word to a control statement on a newline:

```
# WRONG:
if { ... }
{
}
```

```
else
{
}
```

This doesn't work because the "then" code block is on a newline and, therefore, is not seen as an argument to the `if` command. Similarly, the `else` and its code block also are on separate lines and are not seen as part of the `if` command.

2) Using variable substitution when creating a list with curly braces:

```
set a       99
set a_list  { $a 100 101 102 }
```

This doesn't work because no substitutions of any kind are done inside a curly-brace delimited `WORD`.

You also now probably can understand why Tcl has a `set` command and not an assignment statement, since each line needs to start with a command name, so `a = 12` wouldn't fly, as "a" is not a command name.

Because Tcl consists of "commands", another somewhat unexpected thing that you see are "control" statements with options—for example, the `switch` statement looks like this:

```
set var  def
switch $var {
    abc    { puts "won't match this one" }
    def    { puts "should match this one" }
}
```

The `switch` statement also accepts options—for example, the `-glob` option:

```
set var  def
```

```
switch -glob $var {
    abc    { puts "won't match this one" }
    d*f    { puts "should match this one" }
}
```

With the `-glob` option, the matching is done using glob-style matching. Note: Tcl predates the era of double-dash options, so all the standard Tcl commands use single-dash options.

Before wrapping up this section, I want to note one thing about Tcl's variable expansion that should come as a relief to any Bash programmers who have ever been caught in "quote hell": once Tcl expands a variable, it doesn't do any further interpretation of the resulting value.

For example, consider the following Bash code:

```
function one_arg_func()
{
    echo $1
}
```

When executed, you get:

```
$ one_arg_func 1
1
$ one_arg_func "1 2"
1 2
```

That seems fine, but now try this:

```
$ a="1 2"
$ one_arg_func $a
1
```

What happened? Bash happened. It expanded `$a` and then reinterpreted the value as being two separate words and passed two arguments to the function rather than one. Tcl doesn't do that:

```
proc one_arg_func {arg} {
    puts $arg
}

set a  "1 2"
one_arg_func $a
```

Tcl expands the variable `a` once, and that's it, whatever value it has is the value of the argument to the command, regardless of embedded spaces or quotes or anything else that it has in its value.

In *Tcl and the Tk Toolkit*, John Ousterhout, the originator of Tcl, states:

> Tcl's substitutions are simpler and more regular than you may be used to if you've programmed with UNIX shells (particularly csh). When new users run into problems with Tcl substitutions, it is often because they have assumed a more complex model than actually exists.

Some Tcl Commands

The previous examples have contained some Tcl commands; in this section, I want to cover just a few more commands to give you a bit of a feel for what you can do with Tcl, and for how you do it.

Read and write files:

```
# Open and read input file then close the file.
set fd     [open "infile.txt" "r"]
set fdata  [read $fd]
close $fd
```

```
# Open and write output file.
set fd     [open "outfile.txt" "w"]
puts -nonewline $fd $fdata
close $fd
```

Work with strings:

```
# Get length of string.
set len  [string length $str]

# Convert to upper/lower case.
set upper  [string toupper $str]
set upper  [string tolower $str]

# Trim characters (default is spaces).
set trimmed  [string trim $str]

# List of changes to make to a string as a list.
# Values in the first column are changed to the value
# in the second column.
set chgs  {
    abc     def
    ghi     jkl
}
set newstr  [string map $chgs $oldstr]
```

Work with regular expressions:

```
set text {
    My name is Bob
    Hello Bob
    My name is Mary
```

```
    Hello Mary
}

# Find all names found in the phrases "My name is XXX".
set matches  [regexp -nocase -all -inline
 ↪{my\s+name\s+is\s+(\w+)} $text]
foreach {match submatch} $matches {
    puts "Name: $submatch"
}

# Will output:
#   Name: Bob
#   Name: Mary

# Change "My name is XXX" to "Your name is XXX".
set newstr  [regsub -nocase -all {my\s+name\s+is\s+(\w+)}
 ↪$text {Your name is \1}]
puts $newstr

# Will output:
#   Your name is Bob
#   Hello Bob
#   Your name is Mary
#   Hello Mary
```

Work with expressions:

```
set a     1
set sum   [expr 99 + $a]
set lt    [expr $a < 10]
set two   2
set four  [$two << 1]
```

Work with lists:

```
set alist  { 1 2 3 }

# Get first item in list:
set one    [lindex $alist 0]

# Append item to list:
lappend alist 0

# Sort a list:
set slist  [lsort $alist]

# alist is { 1 2 3 0 }
# slist is { 0 1 2 3 }
```

Execute external commands:

```
# Execute grep and put output in Tcl variable:
set result  [exec grep string file.txt]

# Catch errors in execution:
if { [catch {exec grep string file.txt} results options] } {
    puts "Error executing grep"
} else {
    puts "Grep executed ok: $results"
}
```

This only skims the surface of Tcl's commands. For more information on what you can do with Tcl, see the full list of the available Tcl Commands.

The Tk Toolkit

Quite often when you see a reference to Tcl, it's written as Tcl/Tk. Tk is the GUI toolkit

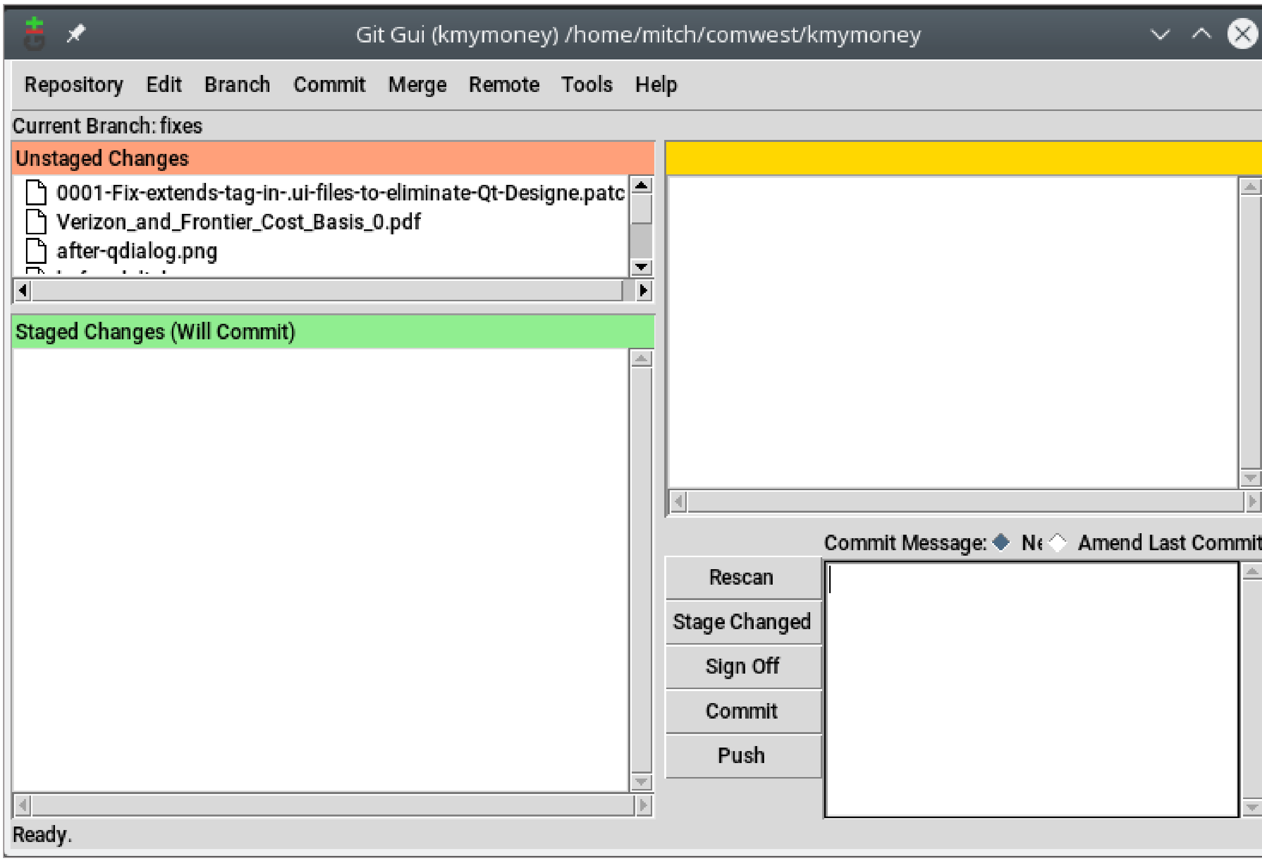


Figure 1. Git GUI

that is closely associated with Tcl, although Tk is usable from other languages (it comes with most Python distributions).

At the start of the article, I noted that the GUI that comes with git is written in Tcl.

So obviously, you can write some pretty sophisticated GUI applications with Tcl/Tk, but if you've ever run the git GUI, you probably weren't exactly "wowed" by its look (unless you long for the days of Motif).

But for some simple GUI tasks, like adding a small pop-up window to some of your command-line scripts, Tk can be quite useful. Note that the latest versions of Tk (in "Tcl-time" this means circa 2007) have themeable widgets, and with a bit of work, you

can make your Tcl/Tk apps look a bit more modern.

As an example of Tk with Tcl, the following code creates a window with two radio buttons and an "ok" button.

When the "ok" button is checked, the "value" of the selected radio button is printed and the program exits:

```
set yesno  -1

wm title . "Which do you like?"
wm geometry . 300x90

radiobutton .rb1 -variable yesno -value 1 -text "I like yes"
radiobutton .rb2 -variable yesno -value 0 -text "I like no"
button .ok -text "Ok" -command { puts $yesno; exit }

grid .rb1 -sticky nw
grid .rb2 -sticky nw
grid .ok
```

In Tk, the top-level window/widget is named `.`, and child widgets are named `.child`. So, for example, a button inside a frame would have a name that looks like ".frame.button".

In the previous example:

- The `wm` command sets some options on the top-level window, the title and the size.

- The `radiobutton .rb1 ...` command creates a radio button named `.rb1`. The radio button displays the text given to the `-text` option, and when it's selected, it sets the variable given to the `-variable` option to the value given to the `-value` option.



Figure 2. Example Window

- And, the same for button `.rb2`.
- The button command creates a push button with the given text, and when the button is pressed, the block argument to the `-command` option is executed.
- The `grid` commands place the buttons into a grid layout (one widget per row in this case). The `-sticky` option sets the widget's alignment.

Valid widget alignment characters are:

- n — North (aka top).
- s — South (aka bottom).
- e — East (aka right).
- w — West (aka left).

Since the script is showing a window, at the end of the script, instead of exiting, the script waits for GUI events. The window should look something like Figure 2.

For more information on what you can do with Tk, see the full list of the available Tk Commands.

Running Tcl

So now that you've seen some Tcl (and a bit of Tk), you're probably itching to try it out.

On Linux, there are two commands for running Tcl scripts: `tclsh` runs scripts that don't use Tk, and `wish` runs scripts that use Tk:

```
$ tclsh my-script.tcl
$ wish my-tkscript.tcl
```

However, note that with many versions of Tcl, you also can use Tk commands with `tclsh` by including the command `package require Tk` in your script before executing any Tk-related commands.

On Linux, Tcl is likely already installed. If not, it should be found in a package named `tcl` in your distro's standard repositories. Another option is to install the commercial distribution of Tcl from ActiveState.

A third option is the open-source Tcl distribution called TclKit. TclKit has some interesting and useful features that I cover a bit more in the next section.

But before I get to that, I need to issue a warning about the next section. I'm going to use some trigger language that may cause discomfort to some readers. So if you're a sensitive type, take a Xanax before reading on.

TclKit—the Batteries-Included Tcl Distribution

TclKits are Tcl and Tcl/Tk distributions that are contained in a single file.

These distributions have a number of uses, and one of the places I've found them to be particularly useful is (and here's the unsettling language) on Windows.

In my day job, I work with Windows, and if you've ever had to write a script (aka batch file) on Windows, you know it can be painful.

Since Windows NT arrived, batch files can do quite a bit more than DOS batch files could, but it's still not very much fun.

I've used both Cygwin and MSYS2, but both of those are heavyweight options that aren't really suited to sending a "quick" script to somebody to run.

I considered using PowerShell, but I haven't yet convinced myself that dotnet for the command line is the way to go, and on top of that, my organization doesn't allow users to run Powershell scripts by default.

So, none of those options work for the sorts of simple tasks I wanted to automate and potentially share with others. And, this is where TclKits come in handy.

A TclKit allows me to write scripts that don't require me to pull my hair out and that I can distribute in a single file, even if the "script" itself is composed of multiple files and includes some additional data files.

And, if I want to distribute it to someone who doesn't even have the TclKit executable, I even can package my scripts into a custom TclKit and still send only one file (albeit a bit bigger than just some script files).

These distributions come in a file that's called a "Starkit". These Starkits contain the Tcl interpreter (optionally with Tk) and any required support files in a virtual filesystem contained inside the executable itself.

Furthermore, you can create your own Starkits and package your scripts and data files inside your custom Starkit and even include a copy of the Tcl interpreter in your Starkit to give yourself a single file executable.

To create your own Starkits, you need to download TclKit (tclkitsh and/or tclkit) and the the Starkit Developer's eXtension (sdx.kit).

To "wrap" your Tcl script into a kit:

```
$ ls
sdx.kit   test.tcl

$ cat test.tcl
puts "Hello Tcl"

# Wrap test.tcl into test.kit.
# Note the Tcl interpreter is not included in the .kit file.
$ tclkitsh sdx.kit qwrap test.tcl
5 updates applied

$ ls
sdx.kit   test.tcl   test.kit
```

The `sdx` command `qwrap` puts your script into a kit, which you can now run:

```
$ tclkitsh test.kit
Hello Tcl
```

Next you can unwrap your kit and see what's inside:

```
$ tclkitsh sdx.kit unwrap test.kit
5 updates applied

$ ls
sdx.kit   test.tcl   test.kit   test.vfs
$ find test.vfs
test.vfs
test.vfs/lib
test.vfs/lib/app-test
test.vfs/lib/app-test/pkgIndex.tcl
test.vfs/lib/app-test/test.tcl
test.vfs/main.tcl
```

The new directory test.vfs contains the unwrapped contents of the kit (vfs stands for Virtual File System).

With an unwrapped Starkit, you now can wrap it into a new Starkit that also includes the Tcl interpreter, giving you a single file executable:

```
$ ls -la test.kit
-rwxr-xr-x ...  781 ... test.kit

# Make copy of tclkitsh to use as the runtime.
$ cp ~/bin/tclkitsh tclkitsh-runtime

# Wrap .vfs and tclkitsh runtime into a single file
$ tclkitsh sdx.kit wrap test -runtime tclkitsh-runtime
4 updates applied
```

This latest wrapping of the script now contains a copy of the Tcl interpreter, so you can run it directly (and distribute it to users that don't have a copy of tclkit):

```
$ ls -la tclkitsh-runtime test
-rwxr-xr-x ... 4421483 ... tclkitsh-runtime
-rwxr-xr-x ... 4425769 ... test

$ ./test
Hello Tcl
```

It's not too exciting so far, but now modify the test script as follows:

```
$ cat test.tcl
package provide app-test 1.0
package require starkit

puts "Hello Tcl"
```

```
set fname  [file join $starkit::topdir payload.txt]
set fd     [open $fname]
set fdata  [read $fd]

puts "Contents of $fname:"
puts $fdata
```

Then create a new file in the VFS and rewrap your Starkit:

```
$ echo "Hello Tcl from VFS" >test.vfs/payload.txt

$ tclkitsh sdx.kit wrap test -runtime tclkitsh-runtime
4 updates applied

$ ./test
Hello Tcl
```

Oops, that didn't do anything. If you look back at the contents of the unwrapped Starkit, you'll notice that there's now a copy of your script in the Starkit. That's the one you need to modify:

```
$ mv test.tcl test.vfs/lib/app-test/

$ tclkitsh sdx.kit wrap test -runtime tclkitsh-runtime
4 updates applied

$ ./test
Hello Tcl
Contents of .../test/payload.txt:
Hello Tcl from VFS
```

So now, you've embedded a data file within your Starkit, opened it from the Tcl script

that's wrapped inside the Starkit and read it like any normal file.

The only difference is the path you used to open it.

Note that on Linux, you can `chmod +x sdx.kit` and put it somewhere in your path and execute it directly.

Conclusion

In this whirlwind tour of Tcl, you've seen what the language looks like, some of the built-in commands, the GUI package named Tk, how to run Tcl scripts and the TclKit version of Tcl. It's lot to digest in one article, but I hope it's given you enough information to get started with Tcl, and I also hope it's garnered enough interest for you to want to give Tcl a try. ■

Mitch Frazier is an embedded systems programmer at Emerson Electric Co. Mitch has been a contributor to and a friend of *Linux Journal* since the early 2000s.

Resources

- Tcl/Tk Website
- ARM
- OpenOCD
- *Tcl and the Tk Toolkit* by John Ousterhout
- Full List of Tcl Commands
- Full List of Tk Commands
- Commercial Distribution of Tcl from ActiveState
- TclKit
- Cygwin
- MSYS2
- PowerShell
- the Starkit Developer's eXtension (sdx.kit)

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

Regular Expressions: the Linux User's Second Language

What are "regular expressions", and why should you bother learning them? This article answers those questions and more.

By Andrew Piziali

Back when I was in high school in the 1970s, if you were taking an advanced placement math class, you could sign up for half an hour of computer time each day. An IBM Selectric typewriter and acoustically coupled modem connected you to an APL\360 timesharing system, ideal for my passion at the time: telescope design using optical ray tracing. Rather quickly I realized that if I was going to be typing throughout my lifetime, touch typing was essential for productivity.

In the same way, if you use a POSIX operating system, such as Linux, regular expressions are essential for productivity—your second language next to glob patterns. In this article, I introduce regular expressions and their origin, distinguish them from glob patterns (the other essential language in a POSIX environment), elaborate on both basic regular expressions (BRE) and extended regular expressions (ERE), and finally discuss one of a number of regular expression extensions. The regular expressions described in this article are based upon the POSIX standard: **IEEE Standard 1003.1-2017**. Now, why do you really



Figure 1. IBM 2741 APL Terminal and Acoustic Modem

need to know this arcane language?

Copy and Paste into a Plain-Text File

Although all of my writing is done in a plain-text editor, such as Vim, I often find myself copying quoted text into the file from a formatted text source, such as a PDF document or web page. Invariably, characters outside the printable character set are pasted in—characters like a left or right quote or an em dash. How can I find those quickly and replace them with a printable equivalent?

The regular expression `[^<tab><sp>-~]`—where `<tab>` is a tab character and `<sp>` is a space character—will match each non-printable character. Hence, using Vim in its normal mode, if I enter:

```
/[^<tab><sp>-~]
```

the cursor will be placed on the next nonprintable character. I can then substitute a printable character at that position. If there are no such characters,

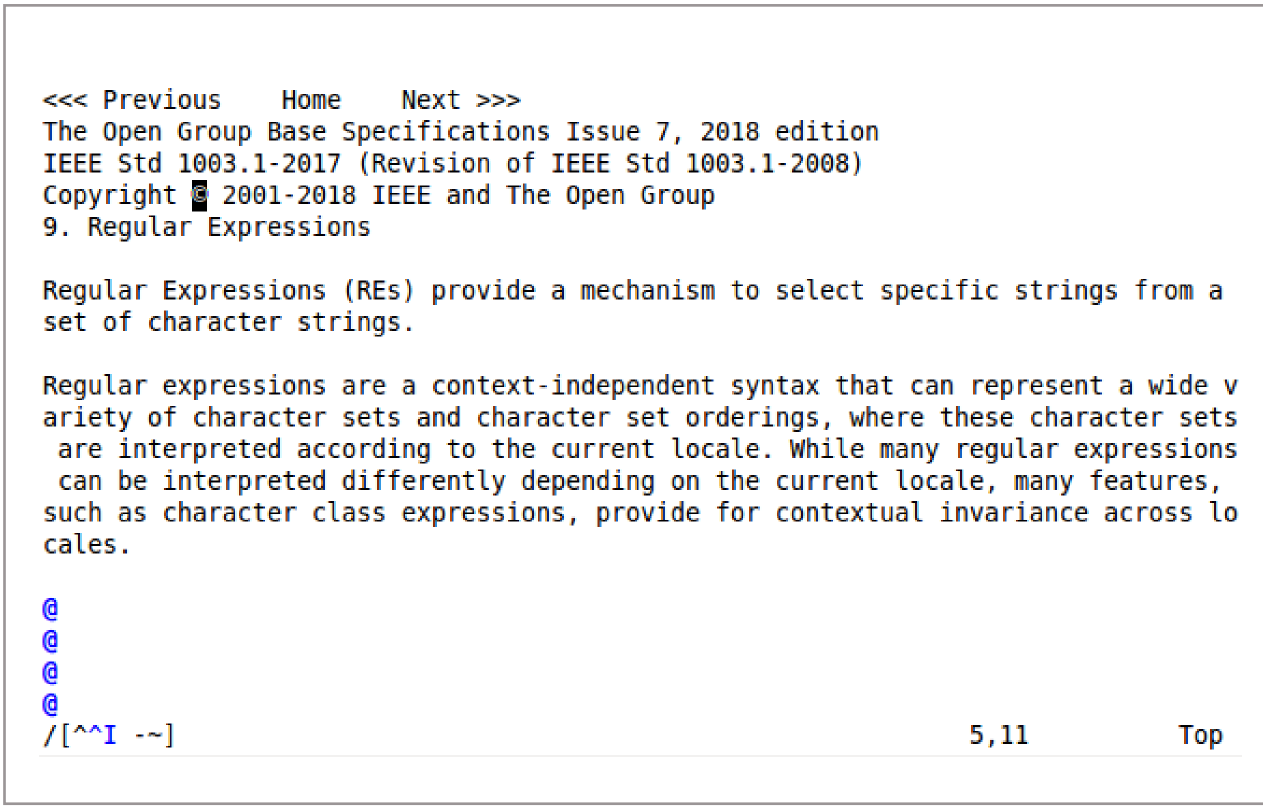


Figure 2. Vim Search for a Non-ASCII Character

Vim will display the message `Pattern not found: [^^I -~].` (I'll revisit this regular expression shortly.)

Throughout this article, I refer to the character as the fundamental unit matched by a regular expression. However, the POSIX standard actually allows for this unit to be a wide character string based upon the current locale. This string is defined as a "collating element".

Before diving into the details of regular expressions, what is this first language of glob patterns?

Glob Patterns

A related, but distinct language from regular expressions in a POSIX environment

is the glob pattern. A glob pattern is most commonly used to find and match file and directory names from a shell prompt. For example, here's the familiar command to list the files that begin with "t" by those files and directories in the current working directory, whose first character is "t" and may have subsequent characters:

```
$ ls -d t*
terry
togo

$
```

The `-d` switch of `ls` tells it to display directory names, not their contents. Hence, in the example above, the shell invokes the program `ls` with the expanded string `ls -d terry togo`. `ls` simply echoes each of its arguments, one line at a time.

As you'll see with regular expressions, glob patterns use ordinary and special characters, but some of those characters have different meanings. If you are typing at a shell prompt, you are using a glob pattern. If you are using a search string in an editor or file browser, such as `less`, you are using a regular expression, sometimes referred to as a "regex".

Origin of Regular Expressions

The American mathematician Stephen Cole Kleene is credited with originating regular expressions when he defined "regular languages". He used a mathematical notation known as a "regular set" to describe a "regular language". A regular set was a forerunner of regular expressions that was used to define the syntax of a programming language.

A year before UNIX was invented (1969), regular expressions were incorporated in the text editor QED by Ken Thompson. At about the same time, these expressions started to be used for lexical analysis by compilers. UNIX Version 1 was introduced with the editor "ed", still a useful tool in a bare-bones Linux

environment. In ed, if you wanted to print those lines in a file that contained the letter pair "re", you would type:

```
g/re/p
```

Look familiar? Yes, the early text search program `grep` took its name from that ed command: `g` (globally), `/` (search), `re` (regular expression), `/` (delimit search), `p` (print). Of course, "re" normally would be a more complex regular expression.

Posix Basic Regular Expressions

The POSIX standard defines two classes of regular expressions: basic and extended. Basic regular expressions (BRE) are derived from Ken Thompson's original use in the ed editor and in `grep`. Extended regular expressions (ERE) were introduced in the 1980s, appearing in the program `egrep`. First I'll dive into basic regular expressions and then extended regular expressions.

Literal Regular Expressions A regular expression (BRE or ERE) is composed of ordinary characters and special characters. An ordinary character simply represents itself. For example "A" and "4" are each ordinary characters, while the special characters ".", "*", "^", "$", "[", "]" and "\" each have an associated regex semantic meaning. If you need to use a special character as an ordinary character, you must precede it with a backslash ("\")—that is, escape it. Hence, the backslash means interpret the following character as an ordinary character, even if it is a backslash itself!

The simplest regular expression is composed solely of ordinary characters. When I typed:

```
g/re/p
```

in the example above, the letter pair "re" is a literal regular expression composed of two ordinary characters. Now let's walk through matching single and multiple characters, using BRE special characters.

Matching a Single Character There are two BREs for matching a single character: the period ("`.`") and the bracket expression. The period matches any character in the character set, except NUL. However, some text processing programs like `grep` further restrict the match to the printable characters, excluding, for example, newline and carriage return. Hence, the regular expression "`.`" matches "a" in the string "abcd" because "a" is *the first* character that matches, starting from the left and scanning to the right.

Suppose though you want to match a single character, and not just any character, but rather one of the characters in a set? BRE defines six bracket expressions for this purpose: the matching list, non-matching list, range, collating symbol, equivalence class and character class expressions. The bracket expression is delimited by the left and right bracket characters: "`[`" and "`]`". (Note the names of these two characters are "left bracket" and "right bracket". They are not "square brackets", because those other two characters—"`{`" and "`}`"—are "left brace" and "right brace". (Stepping off soap box now.) I'll look at each of the bracket expressions in turn, starting with the matching list.

The matching list BRE `[eio]` matches a single character that is one of "e", "i" or "o". Hence, when applied to the string "jettison", it matches the "e", because "e" is the first character that matches a character in the BRE. If applied to the string "archipeligo", it matches the left-most "i".

Sometimes it is easier to specify any character *except* one or more characters. The non-matching list serves this purpose. For example, let's say you want to match a consonant—that is a non-vowel. The BRE `[^aeiou]` performs that match because the caret (`^`) negates the bracket expression. If applied to the string "jettison", it matches the "j". As an aside, the characters period, asterisk, left bracket and backslash are interpreted as ordinary characters within a bracket expression—that is, they are treated as literal characters.

If you want to match one of a set of characters that are consecutive in the character collating sequence, the range operator may be used. `[a-f]` matches

a single letter between "a" and "f". Again, if applied to the string "jettison", it matches the left-most "e". Finally, the basic regular expression defines delimiters for specifying collating symbols, equivalence class expressions and character class expressions. The delimiters are:

- [. .] — collating symbols.
- [= =] — equivalence class expression.
- [: :] — character class expression.

The delimiter pair and its contents match only a single character within a bracket expression.

The symbols used in some locales must be represented by more than one character. For example, the cedilla diacritical mark may appear beneath certain letters, such as "C", yielding the symbol Ç. The corresponding digraph is represented by a letter pair like "ch". The collating symbol expression, such as `[.ch.]`, is used to distinguish the letters that comprise such a pair from the individual letters in a bracket expression. The collating symbol bracket expression `[[.ch.]]` matches a single C cedilla collating symbol; whereas `[ch]` matches the letter "c" or "h". The expression `[A[.ch.]e]` matches the character sequence "A", "Ç", "e".

A locale may define a number of characters that are equivalent to one another. For example, "a", "à" and "á" may be considered the same for the purposes of a regular expression match. The equivalence class expression `[=a=]` defines this equivalence, appearing in a bracket expression as `[[=a=]]`.

A character class expression is a shorthand for a subset of characters in the character class of the current locale. The latter are defined by the value of the environment variable `LC_CTYPE`, such as "en_US.UTF-8". The character class expression is a name delimited by `[:` and `:]`. The following character classes are

supported in all locales:

- `alnum`
- `cntrl`
- `lower`
- `space`
- `alpha`
- `digit`
- `print`
- `upper`
- `blank`
- `graph`
- `punct`
- `xdigit`

Each character class is defined by the `LC_CTYPE` locale category. On my Ubuntu Linux system, the file /usr/share/i18n/locales/i18n defines each class. Hence, if the BRE `[[:digit:]]` is applied against the string "ab4cd5", it will match the third character of the string: "4".

Having explained bracket expressions, the BRE I used in the introduction, `[^<tab><sp>-~]`, can now be understood as "Find the next character that *is not* a tab or a character between space and tilde". Now that you've got matching a single character under your belt, let's look at matching a group of characters.

Matching Multiple Characters Before tackliing multiple character matching, keep in mind a fundamental rule defined by POSIX. If a regular expression will match substrings of various lengths in the examined string, the *longest* string will be matched.

Four expressions are used to match a sequence of multiple characters: concatenation, back reference, zero-or-more repeat and interval repeat. If two BREs are concatenated together, they match the concatenation of their corresponding match strings. For example, the BRE `[A-Z][a-z]` applied to the string "Linux Journal" matches "Li", the

first character pair that is an uppercase letter followed by a lowercase letter.

The back reference expression `\n`, where "n" is a single digit integer 1–9, matches the corresponding nth subexpression of the current BRE enclosed by "\(" and "\)". Hence, the BRE `\(fly\) \1` applied to the string "he fly fly flies" matches "fly fly". "\(" and "\)" may be used to group BRE subexpressions wherever it is convenient.

The zero-or-more repeat is defined by the asterisk operator ("*"). If a BRE is followed by an asterisk, it matches zero or more strings matched by the BRE. The BRE may be a single character, a subexpression or a back reference. If you consider the following string, an H.T. McAdams quote, beginning with:

```
"The problem is
```

and ending with:

```
Lecture Notes," 1967
```

```
"The problem is not that we are ignorant but that we know so much that\n\
isn't true." -- H.T. McAdams, "Elements of Experimental Design -\n\
A Set of Lecture Notes," 1967
```

Table 1 illustrates zero-or-more repeat matches.

Table 1. Zero-or-More Repeat Matches

BRE	Matches
"*The	"The
--*	--
\(we\).*\1	we are ignorant but that we
[[:alpha:]]*	The
i[^i]*i	is not that we are i

Whereas the zero-or-more repeat expression is an all or nothing match, the interval expression allows you to specify precisely the number of BREs that match. When a BRE is followed by one of the following expressions: `\{m\}`, `\{m,\}` or `\{m,n\}`, the string matched by the BRE must be repeated m times, m or more times, or m to n times. m and n are decimal integers, where m is less than n. Note that POSIX disallows specifying "n or less times" using `\{,n\}`, although some applications support this extension.

Referring back to the H.T. McAdams quote above, Table 2 shows the interval expressions matching the illustrated text.

Table 2. Interval Expressions Matching the Example Text

BRE	Matches
\(a[^a]*\)\{2\}	at we are ignor
E[lx][a-z]\{1,\}	Elements
E[lx][a-z]\{1,12\}	Elements of

Anchor Characters If you want to match a character sequence at the beginning or end—or beginning and end—of a line, an anchor character is required. The anchor characters are ^ (circumflex) and $ (dollar sign). When a BRE is preceded by a circumflex, the matching string must consist of the first characters of the examined string. For example, the BRE:

```
^"The problem
```

matches:

```
"The problem
```

in the McAdams string. Likewise, when a BRE is suffixed with a dollar sign, the matching string must be the last characters of the examined string. Again, the BRE `1967$` matches "1967" in McAdams. To anchor the left and right ends of the BRE, use

both the circumflex and the dollar sign. The BRE:

```
^"The problem.*1967$
```

matches the complete McAdams string. Now that I've covered basic regular expressions, let's look at extended regular expressions (EREs).

Posix Extended Regular Expressions

The most ubiquitous application of extended regular expressions is in the program egrep, a deprecated means of invoking grep with the -E switch. ERE adds three special characters—?, + and |—and eliminates the need to escape ( ) and { }. In other words, the latter four also become special characters. I'll describe each in turn.

Just as an asterisk matches the previous BRE zero or more times, the question mark matches the previous ERE once or not at all. Hence, the ERE (is)? not applied to the McAdams string matches "is not".

Similarly, the plus sign matches the preceding ERE one or more times. This means the ERE (that we[a-z]+)+ matches:

```
that we are ignorant but that we know so much that
```

in the McAdams string.

I find the vertical bar ("|") operator one of the most useful, because it extends the OR function of single character matching in a bracket expression to an OR of arbitrary EREs. For example, (The|that) matches (surprise!) "The" and "that" in the McAdams string. Yet, an ERE as complex as <sp>(th|kn|tr)[[:alpha:]]* also matches the words "that", "know" and "true" in the McAdams string.

The brace characters { and } serve the same purpose for an interval expression as in a BRE, without requiring a preceding backslash. Hence, [1-9]{1,4} matches "1967" in the McAdams string.

Regular Expression Extensions

In addition to POSIX-defined basic and extended regular expressions, a variety of extensions to regular expressions exist in the wild, such as those in Henry Spencer's regex library, Apache, Tcl and Perl. However, the most widely used regular expression extensions are those defined by Perl. I briefly touch on them here, but the definitive reference is perldoc.perl.org.

Perl regular expressions (PREs) build upon EREs in a number of ways, including extended escape sequences, ERE modifiers, extended bracketed character classes and zero-width assertions. Let's look at each of these.

Although ERE restricts escape sequences to \1–\9, PRE allows any ASCII letter or digit to either represent a character or serve another purpose. For example, `\a`, `\e` and `\n` represent the bell, escape and newline characters. `\N{...}` matches a named or numbered Unicode character or sequence.

PRE allows you to change the default behavior for matching using modifiers. A modifier is a letter appended to a Perl pattern, such as `/ab\ncd/m`. The "m" means treat the string as multiple lines so that the anchors "^" and "$" match the beginning and end of string, rather than the beginning and end of line. `/ab\ncd/m` matches the five characters "a", "b", newline, "c" and "d".

An extended bracketed character class is a POSIX bracket expression that supports more readable classes and set operations. It uses the syntax `?[...]`, where the "..." is a character class set expression. For example, the expression `?[ \p{Thai} & \p{Digit} ]` matches any Thai digit character.

Zero-width assertions match the context of characters rather than characters themselves. For example, `\b` matches a boundary between a word and non-word.

Conclusion

In this article, I surveyed regular expressions—both basic and extended—and discussed their history. I distinguished them from glob patterns, which are used

primarily at the command-line shell prompt. Finally, I touched briefly on Perl regular expressions, the most common of a number of extensions. If you become proficient using this second language of POSIX computing, you'll find navigating through plain-text files far easier and more efficient! ■

Andrew Piziali cut his teeth on APL\360 in high school, maintained COBOL, FORTRAN and BPL compilers for the US Air Force and fell in love with Pascal running on a PDP 11/70 near the end of his enlistment. After receiving a BSEE in 1983, he pursued a career in function design verification. Andrew installed IBM Xenix 3.0 on his IBM PC-AT in 1984. He replaced this with Softlanding Linux System (SLS) in 1992, based on the Linux 0.95 kernel, and he has had a Linux-only household ever since. He applied his engineering expertise for 25 years, verifying mainframes, supercomputers and microprocessors. Having an avid interest in coverage-driven verification, in 2004 he authored the book *Functional Verification Coverage Measurement and Analysis*. Later he co-authored *ESL Design and Verification* with Grant Martin and Brian Bailey.

Resources

- Regular-Expressions.info
- Regular Expression History (Wikipedia)
- Pattern Matching Notation
- Regular Expression Definitions
- Printable Character Set
- perlre

The Best Command-Line-Only Video Games

A rundown of the biggest, most expansive and impressive games that you can run entirely in your Linux shell.

By Bryan Lunduke

The original UNIX operating system was created, in large part, to facilitate porting a video game to a different computer. And, without UNIX, we wouldn't have Linux, which means we owe the very existence of Linux to...video games.

It's crazy, but it's true.

With that in mind, and in celebration of all things shell/terminal/command line, I want to introduce some of the best video games that run entirely in a shell—no graphics, just ASCII jumping around the screen.

And, when I say "best", I mean the *very* best—the terminal games that really stand out above the rest.

Although these games may not be considered to have "modern fancy-pants graphics" (also known as MFPG—it's a technical term), they are fantastically fun. Some are big, sprawling adventures, and others are smaller time-wasters. Either way, none of them are terribly large (in terms of drive storage space), and they deserve a place on any Linux rig.

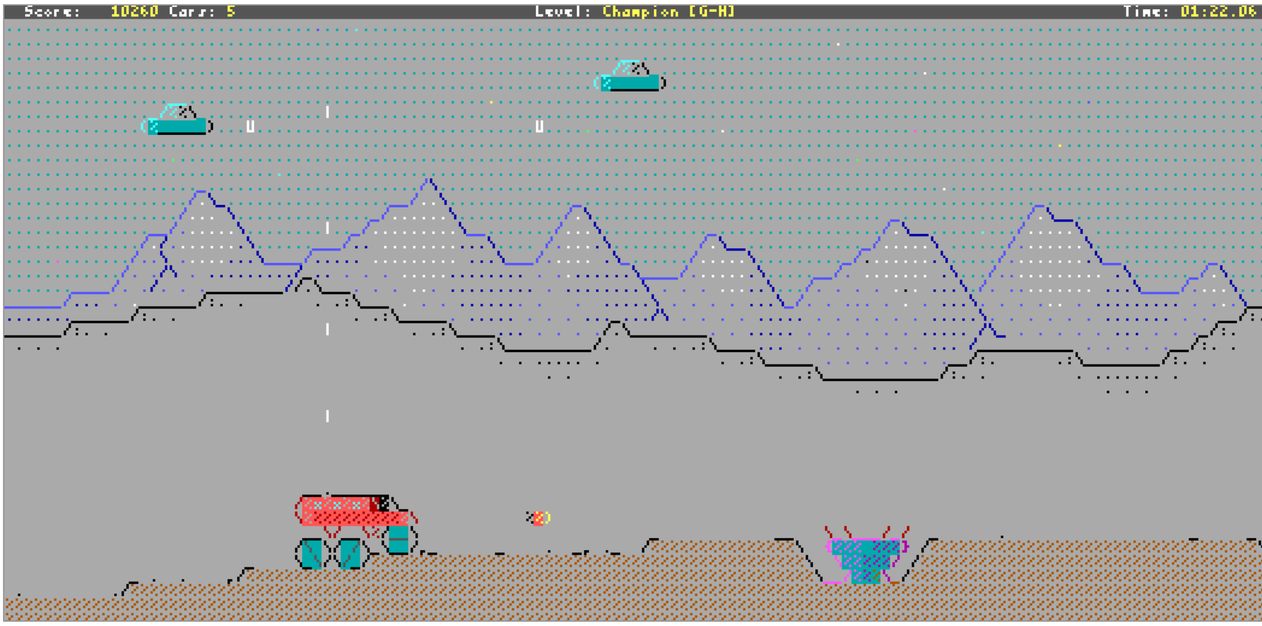


Figure 1. Shooting Aliens and Dodging Potholes in *AsciiPatrol*

AsciiPatrol *AsciiPatrol* is, in my opinion, one of the most impressive terminal games out there. A clone of the classic *Moon Patrol*, which is a ton of fun already, this terminal-based game provides surprisingly good visuals for a game using only ASCII characters for artwork.

It has color, parallax scrolling backgrounds, animated enemies, sound effects—I mean, even the opening screen is impressive looking in a terminal.

For a quick round of arcade-style fun, this one really can't be beat.

Cataclysm: Dark Days Ahead *Cataclysm: Dark Days Ahead* is absolutely huge in scale. Think of it as a top-down, rogue-like, survival game with zombies, monsters and real end-of-the-world-type stuff.

The game features a crafting system, bodily injuries (such as a broken arm), bionic implants, farming, building of structures and vehicles, a huge map (with destructible terrain)—this game is massive. The visuals may be incredibly simple,

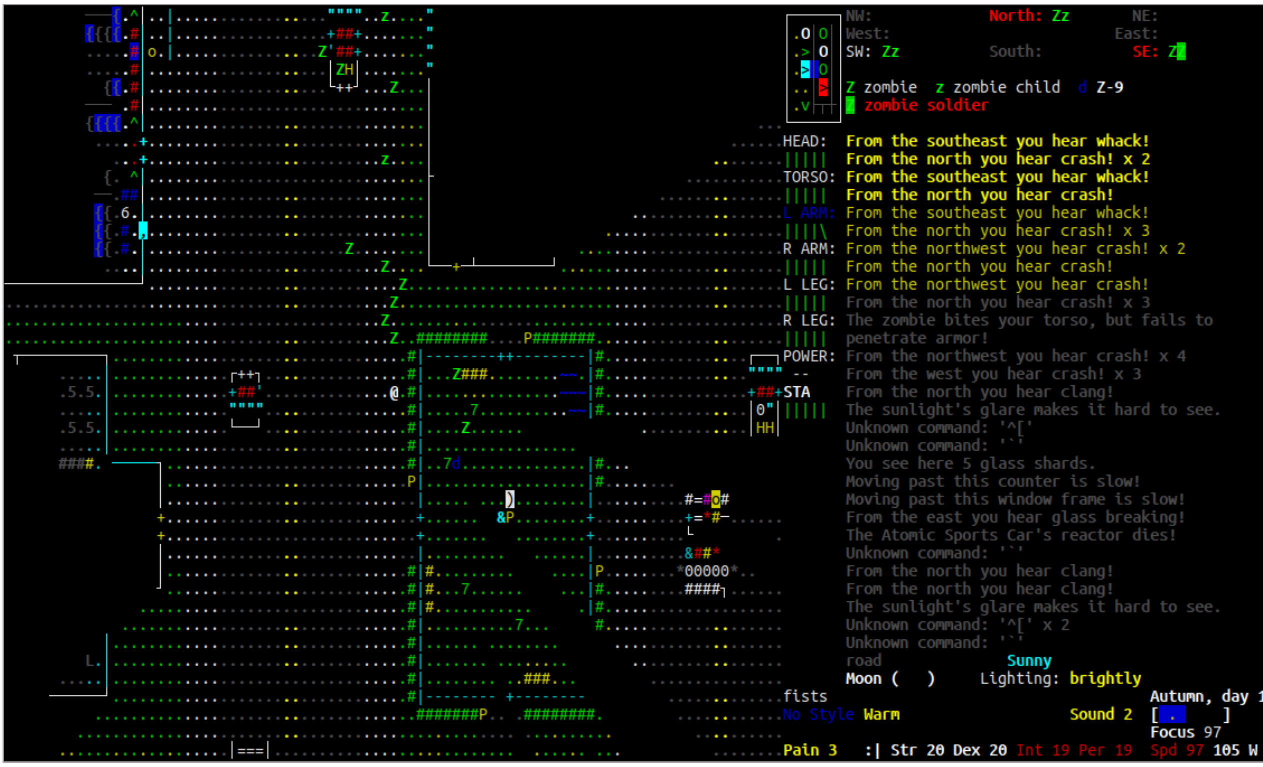


Figure 2. Running from zombies in *Cataclysm*

but the gameplay is deep and open-ended.

SSHTron The *Tron*-inspired light-cycle games (and non-*Tron*-themed variants, such as *Snake*) have been a staple of gaming since the 1980s. And, *SSHTron* provides a four-player version right in your terminal.

Simply open your terminal and type in the following:

```
ssh sshtron.zachlatta.com
```

And, away you go! You'll instantly be connected and can join a game with up to three other players. It's simple. It's quick. It's fun. You can't beat that.

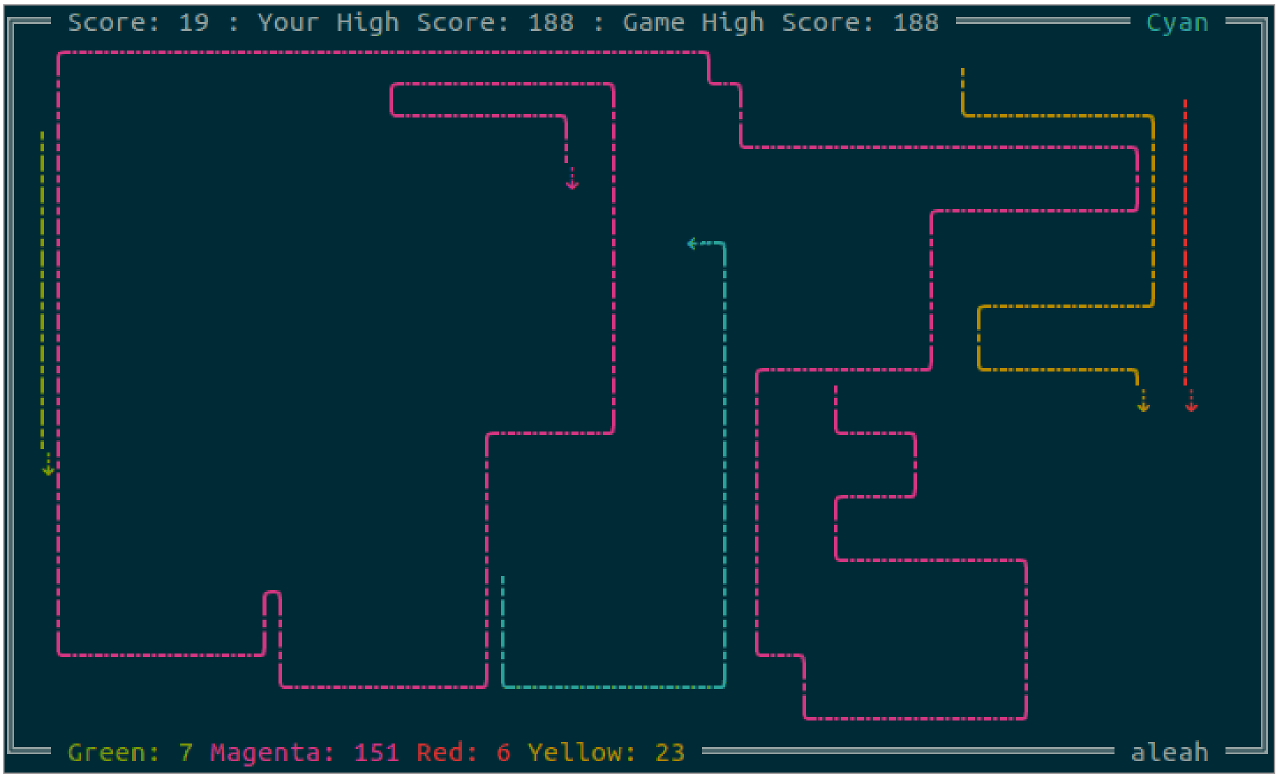


Figure 3. Magenta FTW in *SSHTron*



Figure 4. Find your very own missile launcher in *DoomRL*.

DRL (Doom, Rogue-Like) What if you took the classic first-person shooter, *Doom*, and turned it into a top-down, dungeon-crawling adventure (à la the classic *Rogue*)? Enter *DRM* (aka *Doom*...Rogue-Like).

The gameplay is fast and easy to pick up. It's a quick way to get your adventure game fix in without spending a huge amount of time playing something more demanding (like *Cataclysm*).

Ascii Sector This is one of my personal favorites.

Ascii Sector is a space-exploration game set entirely in your terminal. Travel around between worlds, trade goods, fight alien ships, upgrade your ship, go on quests. The scope is huge, and the atmosphere is delightfully retro-sci-fi.



Figure 5. Approaching a Planet in *Ascii Sector*

Think of this game as being in the same mold as *Elite, Wing Commander: Privateer* or *TradeWars 2002*. If you've ever enjoyed any of those, *Ascii Sector* will not disappoint.

There are moments in this game that simply make me smile. When coming to a planet, for example, and it's displayed entirely in colorful ASCII, it just looks glorious. I can't recommend this game enough.

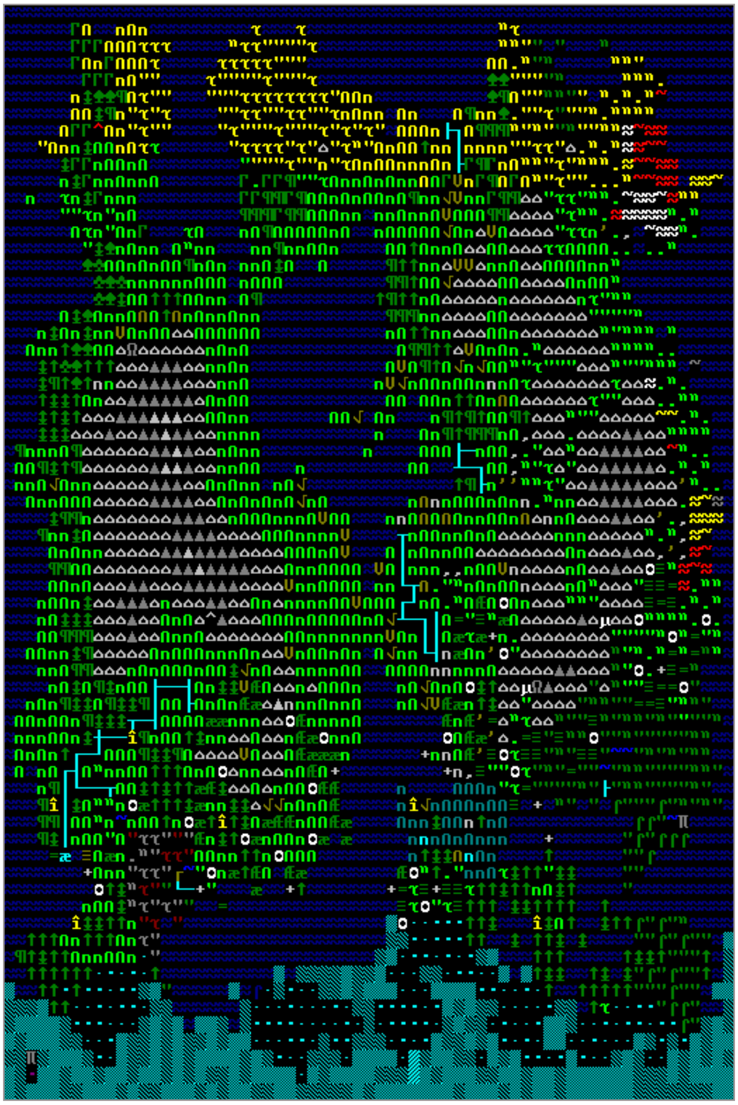


Figure 6. One of Many World Maps in *Dwarf Fortress*

Dwarf Fortress This is the only non-open-source game I'm recommending on this list. But the game is so truly spectacular, it has earned a place here. And, it's free (as in beer).

Think of *Dwarf Fortress* like a combination between *Minecraft*, a top-down adventure game and a general construction simulator. There's a huge world to explore and build, with the player not so much directly controlling any of the characters, as giving them tasks and roles. Woodworking, crafting, farming, brewing—there are so many details and options in this game.

Dwarf Fortress is the kind of game you easily can sink countless hours (and days and weeks) into. It's absolutely staggering in scope and complexity.

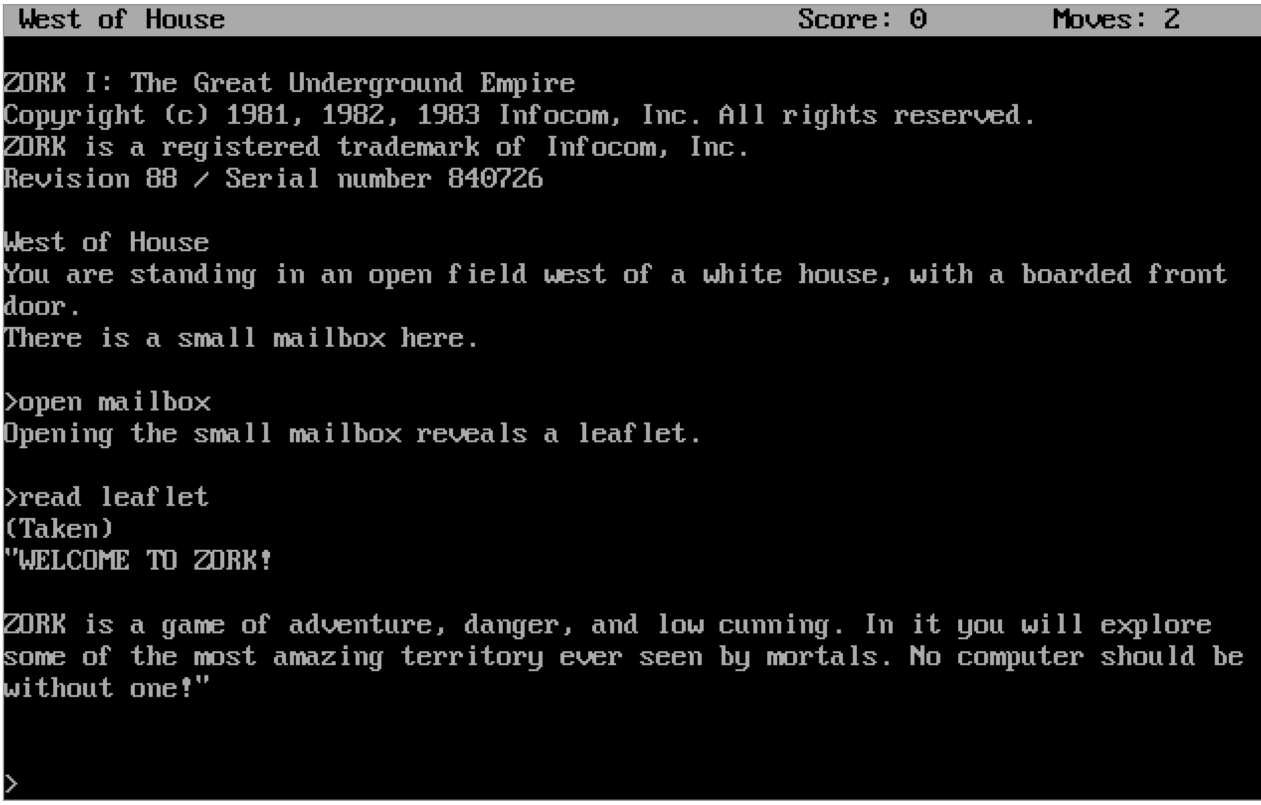


Figure 7. At least you haven't been eaten by a grue yet.

Frotz Although not technically a "game" in the traditional sense of the word, *Frotz* is an interpreter for text adventure games—like the Infocom classic, *Zork*.

Being able to enjoy these adventures, many of which still hold up today, right in your terminal is absolutely delightful. You can find text adventures (or interactive fiction) all over the internet. Some made by companies long since abandoned, others released (usually for free) by independent creators.

Making the Experience Fancier

If you really want to get the most out of playing games in your terminal, you'll want to make sure you have a terminal emulator that does them justice—especially one that supports color text.

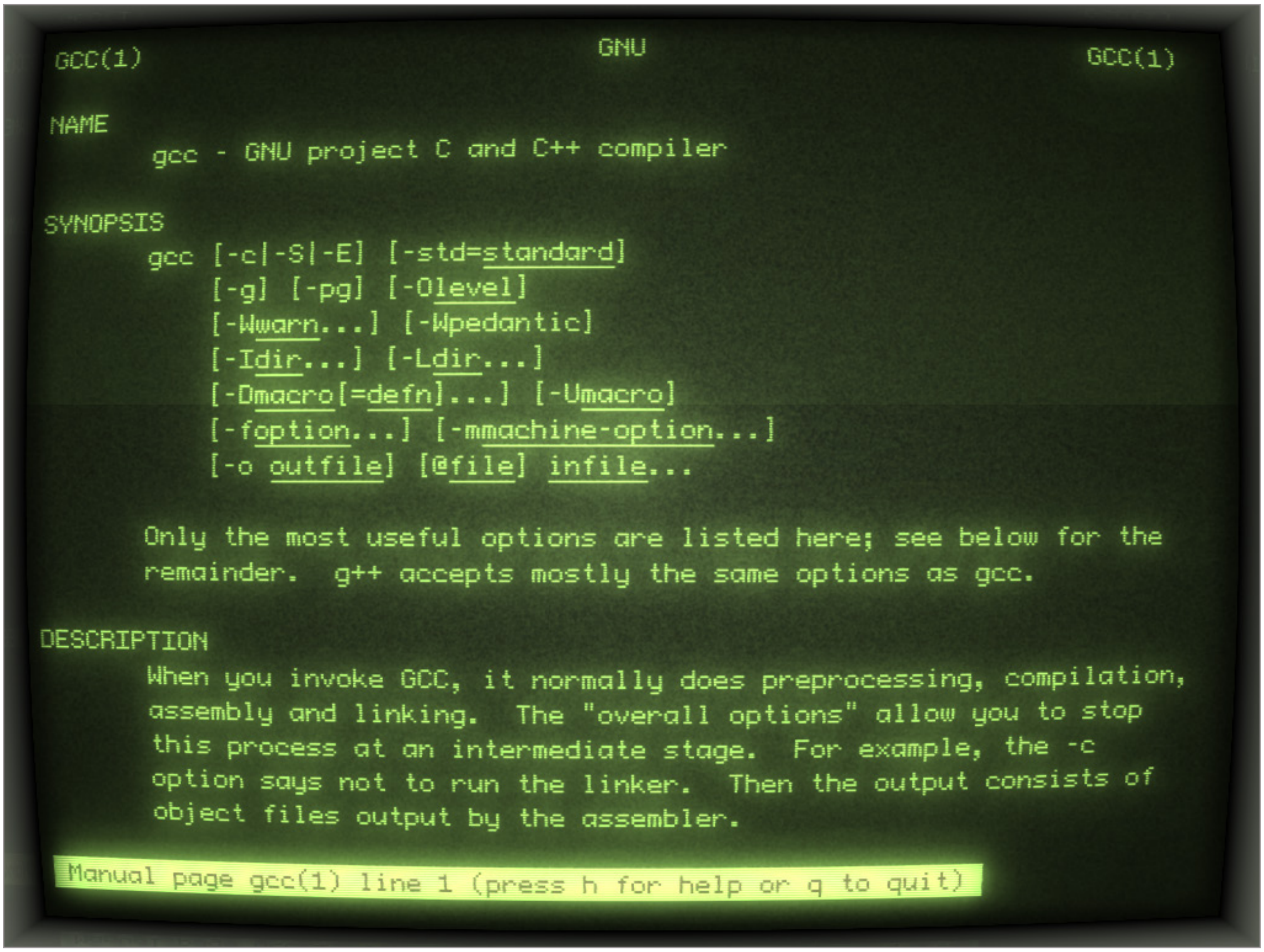


Figure 8. Even the man page for GCC looks fancy in Cool-Retro-Term.

And, if you really want to step up your "I'm playing these like they were in the 1980s" game, I recommend cool-retro-term. It's a terminal emulator that mimics (quite well) the look of old CRT monitors, including tons of options for scan lines, amber text and more. It really makes these games pop.

Cool-Retro-Term is sure to win over even the most die-hard skeptic of playing text-based games. ■



Bryan Lunduke is a former Software Tester, former Programmer, former VP of Technology, former Linux Marketing Guy (tm), former openSUSE Board Member... and current Deputy Editor of *Linux Journal*, Marketing Director for Purism, as well as host of the popular *Lunduke Show*. More details: http://lunduke.com.

Resources

- AsciiPatrol
- Cataclysm: Dark Days Ahead
- SSHTron
- DoomRL
- Ascii Sector
- Dwarf Fortress
- Frotz
- Cool-Retro-Term

Send comments or feedback via http://www.linuxjournal.com/contact or email ljeditor@linuxjournal.com.

An AI Wizard of Words

A look at using OpenAI's Generative Pretrained Transformer 2 (GPT-2) to generate text.

By Marcel Gagné

It's probably fair to say that there's more than one person out there who is worried about some version of artificial intelligence, or AI, possibly in a robot body of some kind, taking people's jobs. Anything that is repetitive or easily described is considered fair game for a robot, so driving a car or working in a factory is fair game.

Until recently, we could tell ourselves that people like yours truly—the writers and those who create things using some form of creativity—were more or less immune to the march of the machines. Then came GPT-2, which stands for Generative Pretrained Transformer 2. I think you'll agree, that isn't the sexiest name imaginable for a civilization-ending text bot. And since it's version 2, I imagine that like *Star Trek*'s M-5 computer, perhaps GPT-1 wasn't entirely successful. That would be the original series episode titled, "The Ultimate Computer", if you want to check it out.

So what does the name "GPT-2" stand for? Well, "generative" means pretty much what it sounds like. The program generates text based on a predictive model, much like your phone suggests the next word as you type. The "pretrained" part is also quite obvious in that the model released by OpenAI has been built and fine-tuned for a specific purpose. The last word, "Transformer", refers to the "transformer architecture", which is a neural network design architecture suited for understanding language. If you want to dig deeper into that last one, I've included a link from a Google AI blog that compares it to other machine learning architecture (see Resources).

On February 14, 2019, Valentine's Day, OpenAI released GPT-2 with a warning:

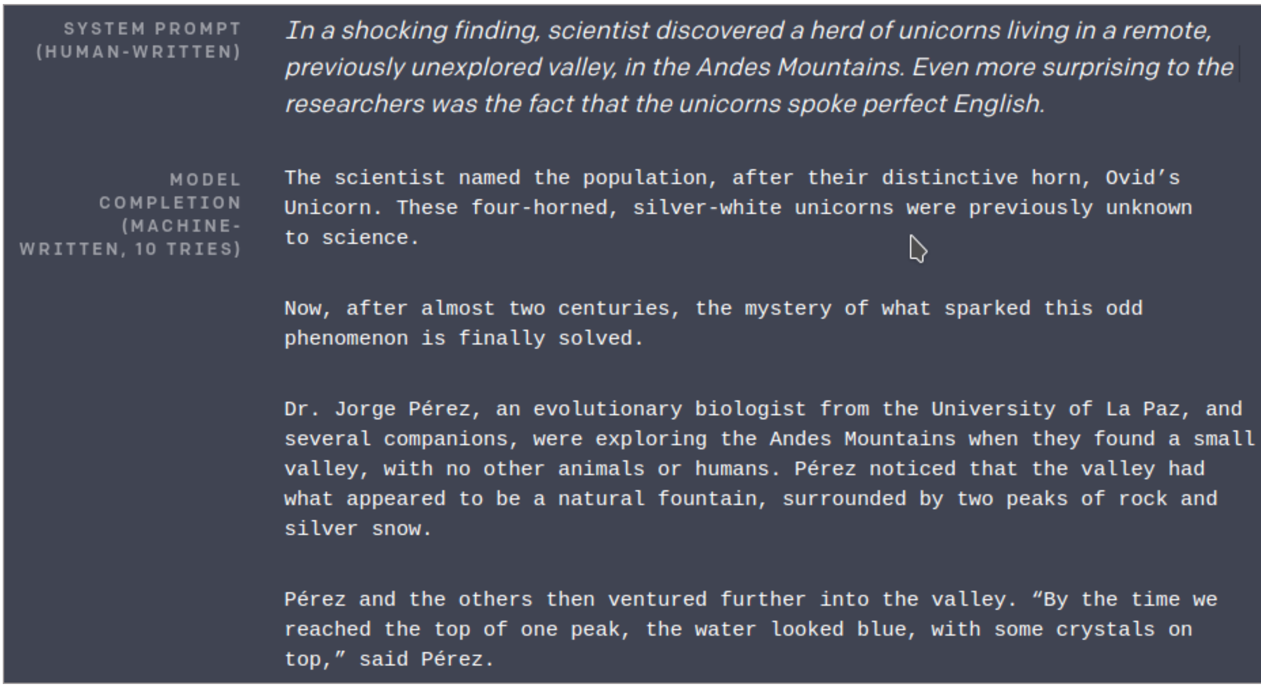
> Our model, called GPT-2 (a successor to GPT), was trained simply to predict the next word in 40GB of Internet text. Due to our concerns about malicious applications of the technology, we are not releasing the trained model. As an experiment in responsible disclosure, we are instead releasing a much smaller model for researchers to experiment with, as well as a technical paper.

I've included a link to the blog in the Resources section at the end of this article. It's worth reading partly because it demonstrates a sample of what this software is capable of using the full model (see Figure 1 for a sample). We already have a problem with human-generated fake news; imagine a tireless machine capable of churning out vast quantities of news and posting it all over the internet, and you start to get a feel for the dangers. For that reason, OpenAI released a much smaller model to demonstrate its capabilities and to engage researchers and developers.

If you want to try this "too dangerous to release" AI for yourself, you can. Here's what you need to do. OpenAI has a GitHub page for the GPT-2 code from which you can either download via a git clone or simply pick up the latest bundle as a ZIP file:

```
$ git clone https://github.com/openai/gpt-2.git
Cloning into 'gpt-2'...
remote: Enumerating objects: 174, done.
remote: Total 174 (delta 0), reused 0 (delta 0), pack-reused 174
Receiving objects: 100% (174/174), 4.35 MiB | 1.72 MiB/s, done.
Resolving deltas: 100% (89/89), done.
```

This will create a folder called "gpt-2" from which everything else will flow. Before you can jump in and make this all work, you're likely going to need to install a few prerequisites. The biggest of these is a Python 3 environment, pip and tqdm. If you are lucky enough to have an NVIDIA GPU on-board, you'll also want to install CUDA; it's not required, but it does make things go a lot faster. On my Ubuntu system, I



SYSTEM PROMPT (HUMAN-WRITTEN)

In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

MODEL COMPLETION (MACHINE-WRITTEN, 10 TRIES)

The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science.

Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved.

Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow.

Pérez and the others then ventured further into the valley. "By the time we reached the top of one peak, the water looked blue, with some crystals on top," said Pérez.

Figure 1. Part of the Sample Provided in the OpenAI Blog

installed the packages like this:

```
sudo apt install python3-pip python3-tqdm python3-cuda
```

Before I continue, here's a treat. When the code was first released, it included a 117M (million) parameter model to limit the potential danger of releasing a better version into the wild. Apparently, some of those fears have been put to rest, because as of May 4, 2019, there is now a 345M parameter model. The largest model in the code base, if and when it is released, is (or will be) 1542M parameters.

You will need that 345M model on your computer, so let's download it now:

```
python3 download_model.py 345M
```

How long this step takes will depend a bit on your connection speed since you are downloading a lot of data, so this might be a good time to get yourself a snack,

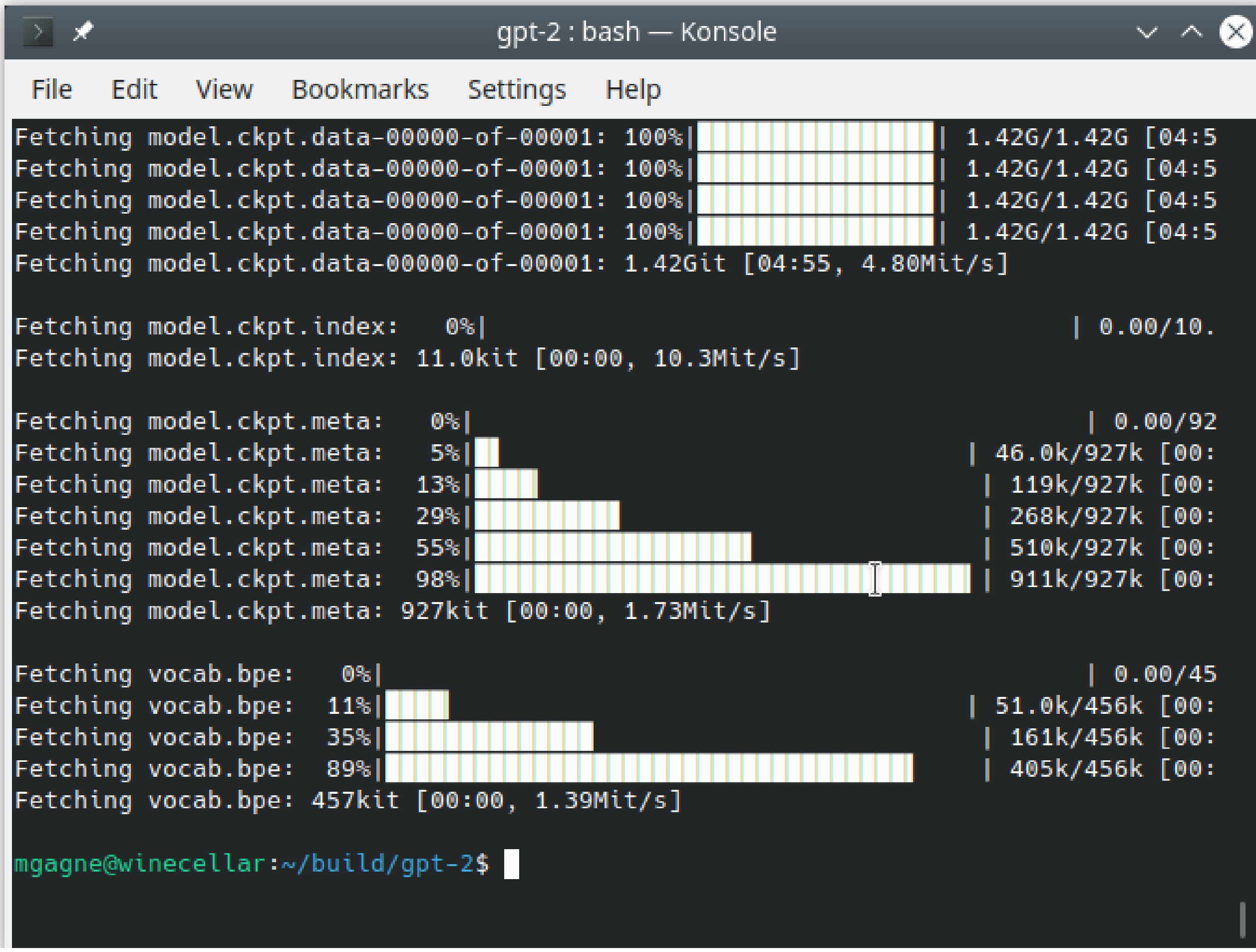


Figure 2. Downloading the GPT-2 Language Model

or a drink. As it downloads, you'll get a visual update on the various parts of the model (Figure 2).

Yes, there are more prerequisites to install. Luckily, you can install a number of them via a file in the gpt-2 source called requirements.txt:

```
pip3 install -r requirements.txt
```

There are only four packages, so you also can just do this:

```
pip3 install fire regex requests tqdm
```

The next step is to install tensorflow, of which there are two versions. GPT-2 will run on any system where the requirements are met, but if you happen to be lucky enough to have that NVIDIA GPU with the appropriate driver, it will all run much faster. To install the GPU version of the tensorflow code, do the following:

```
pip3 install tensorflow-gpu==1.12.0
```

To install the non-GPU version, the command looks like this:

```
pip3 install tensorflow-gpu==1.12.0
```

Once again, this is a big package, so it might take a few minutes. Once done, several other packages, all of which make up tensorflow, also will have been installed. By this point, a number of commands will have been installed in your $HOME directory under .local/bin. Save yourself some pain and include that in your .bash_profile's `$PATH`. If you're in a hurry, and you don't want to log out and back in right now, you can always update your `$PATH` on the fly:

```
export PATH=$PATH:/home/mgagne/.local/bin
```

Perfect! Now you're ready to generate a textual masterpiece. In the src directory, you'll see two scripts to generate text:

```
src/generate_unconditional_samples.py
src/interactive_conditional_samples.py
```

Let's start with the unconditional script.

```
python3 src/generate_unconditional_samples.py --top_k 40
 ↪--model-name 345M
```

If you are running this entirely from the CPU, it may take a few seconds to start generating text, so be patient. Before I show you a sample of what I managed to

generate on my first pass, I want you to look at a couple command-line options. One of them is fairly obvious, and that's the `--model-name` option, which selects the model should you have more than one installed. Remember that there are now two available; one is 117M parameters, and the other is 345M. The second option I want you to look at is `top_k`, which represents the percentage of logits used in selecting words. A lower value will tend to create text using simpler words, but it also tends to be more repetitive. A higher `top_k` tends to generate more realistic text.

To make things more interesting, you'll want to give the AI writer a place to start by providing it an opening line, and that means you'll want to use the interactive script:

```
python3 src/interactive_conditional_samples.py --top_k 40
 ↪--model-name 345M
```

When you run this, it will take a few seconds (or more if you are using the CPU version of tensorflow), at which point it will give you a `Model prompt >>>` prompt. This is where you type in your line of text. When you press Enter, the magic begins. Again, be patient. I decided to give it a simple one-line prompt of "Once upon a time, there was a beautiful princess." Everyone likes a fairy tale, right?

With a `top_k` of 10, the AI writer produced the following (I'm including only the first two paragraphs):

> The princess was a princess of the land of the moon. The moon was her favorite. She was born a princess, but she was not a princess in her own right. She was called a princess by the people of the moon, but they didn't know what it meant or what it meant to be a princess.
>
> The princess had the same name as her father.

Let's try dropping that `top_k` to 1 using the same prompt. I'll show you the first five paragraphs because they're so short:

> She had a beautiful face, a beautiful smile.
>
> Her eyes, her hair, her body were all beautiful.
>
> Her hair was long and beautiful.
>
> Her eyes, her hair, her body, her body was all beautiful.
>
> Her hair was long and beautiful.

Ouch! I think the books my kids read when they were five years old had more depth.

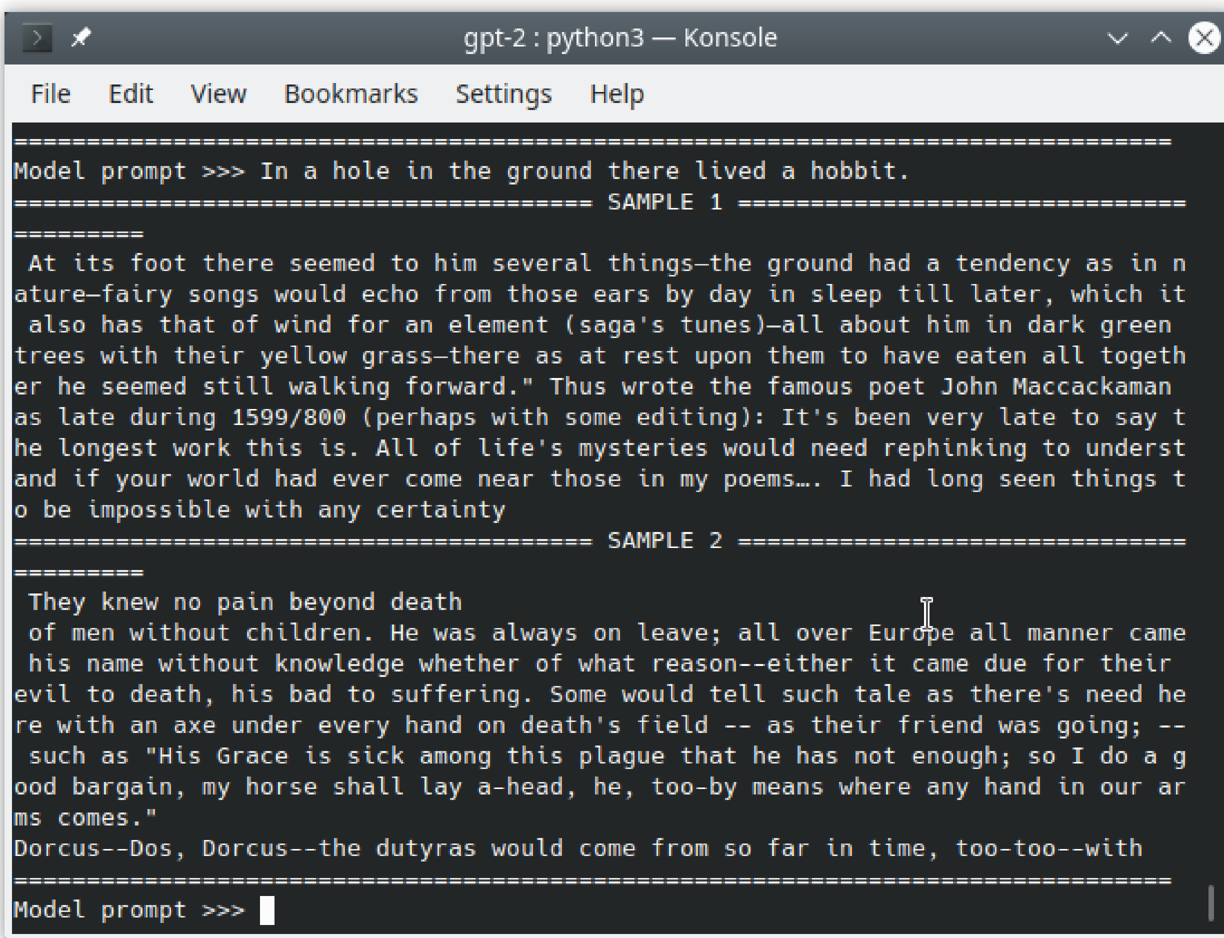


Figure 3. Tolkien is safe—for now.

Okay, one last pass before visiting some of the other command-line options you can use. I've gone back to the default of a `top_k` of 40:

> At the age of seven, she learned to read and write, became a famous author, gave birth to a handsome prince (in other words, she was just about to enter the Royal Family). At age twenty she reached puberty. At seventeen she gave birth to a prince of the same class as her own. At twenty-four she had already entered her second period, and was then named Countess. What happened to her?
>
> That's when I realized that I was living a story, that was told on both sides of every age-related phenomenon. There's always a second gender, a third gender, a fourth gender. And when someone falls into any of those different, the gender they are inside of can change in accordance with their situation. As it turns out, the "male" aspect in life may actually be the opposite of the "female" aspect of reality.

So, fiction writers are probably okay for a little while, or at least until OpenAI releases the full model, but it does give you an interesting example of where this is all going.

Some of other command-line options hidden in the code include the following:

- `--nsamples`
- `--length`
- `--temperature`

I've been experimenting with these to see how they affect results. Temperature is interesting in that it affects the "creativity" of the program in that it decreases the likelihood that the AI writer will take the safe road. If you want more than one sample generated each time, set the `--nsamples` number. The `--length` option is measured in words. So, let's say I want two samples of 150 words each, I might issue the following command (note that the whole command is actually one line):

```
python3 src/interactive_conditional_samples.py --top_k 40
 ↪--temperature 5 --length 150 --nsamples 2
 ↪--model-name 345M
```

As a starting point, I used the first line of Tolkien's *The Hobbit*: "In a hole in the ground there lived a hobbit." Figure 3 shows the results. Let's just say that Tolkien, if he were still alive, would be safe as a novelist—for now.

For a more complete list of command-line options to GPT-2, use the following command (pay attention to the two hyphens before the `--help`):

```
python3 src/interactive_conditional_samples.py -- --help
```

I've been making fun of the output, but there's something amazing happening here that can't be completely ignored. The model I'm forced to work with is better than the original that was released, but it's nowhere near what OpenAI still has sitting out there. This also is still in development, so it has a way to go. In addition, as I mentioned in the opening, there is a dark side to this that can't be ignored, beyond even the potential career-ending aspect that writers like myself may be facing, and that's the spectre of just what these tireless AI writers may be releasing into the world.

One sample I did not include in this article made me cringe with horror.

I submitted the first line of Jane Austen's *Pride and Prejudice*: "It is a truth universally acknowledged, that a single man in possession of a good fortune, must be in want of a wife."

The result was hateful, misogynistic, homophobic and included made-up but plausible quotes from the *Bible*. This is fascinating technology and well worth your time and exploration. The more we all understand these forays into machine learning and artificial intelligence, the better prepared we will be when Jane Austen turns into Alex Jones. ■

Marcel Gagné is Writer and Free Thinker at Large. The Cooking With Linux guy. Ruggedly handsome! Science, Linux and technology geek. Occasionally opinionated. Always confused. Loves wine, food, music and the occasional single malt Scotch.

Resources

- Google AI Blog: Transformer: A Novel Neural Network Architecture for Language Understanding
- OpenAI GPT-2 on GitHub
- OpenAI Blog: Better Language Models and Their Implications
- YouTube clip from Star Trek's "The Ultimate Computer"

Linux IoT Development: Adjusting from a Binary OS to the Yocto Project Workflow

Introducing the Yocto Project and the benefits of using it in embedded Linux development.

By Mirza Krak

In embedded Linux development, there are two approaches when it comes to what operating system to run on your device. You either build your own distribution (with tools such as Yocto/OpenEmbedded-Core, Buildroot and so on), or you use a binary distribution where Debian and derivatives are common.

It's common to start out with a binary distribution. This is a natural approach, because it's a familiar environment for most people who have used Linux on a PC. All the commodities are in place, and someone else has created the distribution image for you to download. There normally are custom vendor images for specific hardware that contain optimizations to make it easy to get started to utilize your hardware fully.

Any package imaginable is an `apt install` command away. This, of course, makes it suitable for prototyping and evaluation, giving you a head start in developing your application and your product. In some cases, you even might ship pre-series devices using this setup to evaluate your idea and product further. This is referred to as the "golden image" approach and involves the

following steps:

1. Flash the downloaded Debian image to an SD card.

2. Boot the SD card, log in and make any modifications needed (for example, installing custom applications). Once all the modifications are complete, this becomes your golden image.

3. Duplicate the SD card into an image on your workstation (for example, using `dd`).

4. Flash the "golden image" to a fleet of devices.

And every time you need to make a change, you just repeat steps 2–4, with one change—that is, you boot the already saved "golden image" in step 2 instead of the "vanilla" image.

At a certain point, the approach of downloading a pre-built distribution image and applying changes to it manually will become a problem, as it does not scale well and is error-prone due to the amount of manual labor that can lead to inconsistent output. The optimization would be to find ways to automate this, generating distribution images that contain your applications and your configuration in a reproducible way.

This is a crossroad where you decide either to stick with a binary distribution or move your idea and the result of the evaluation and prototyping phase to a tool that's able to generate custom distributions and images in a reproducible and automated way.

There are, of course, ways to generate custom Debian images, but the problem here is fragmentation. If you're using vendor-provided images, they probably have created their own tools (a bash script wrapper around debootstrap) to generate those images that you might be able to get access to. The fragmentation results

in very little re-use, and if you decide to change the hardware later but still base it on Debian, you might need to re-work your process completely, as this might be using a different set of tools.

The remainder of this article assumes you've made the choice to switch to a tool that's able to build Linux distributions—specifically the Yocto Project, which is based on OpenEmbedded-Core. This has some implications, and I try to cover the key parts.

Here's a quote from the yoctoproject.org website to give you a quick summary of the Yocto Project:

> The Yocto Project (YP) is an open-source collaboration project that helps developers create custom Linux-based systems regardless of the hardware architecture.
>
> The project provides a flexible set of tools and a space where embedded developers worldwide can share technologies, software stacks, configurations, and best practices that can be used to create tailored Linux images for embedded and IOT devices, or anywhere a customized Linux OS is needed.

Next, let's look at the key differences when moving from a binary distribution to the Yocto Project that will impact your workflow.

The Yocto Project Is a Cross-Development Environment

Because the Yocto Project is a cross-development environment, this means the build and generation of the custom Linux distribution image happens on the host machine, with the intention of the output running on a target. This could mean that the host is an x86_64 machine and the target is an ARM-v7—hence, the "cross-development".

And, this can be frightening at first; coming from a binary distribution, you might

not have encountered this workflow before. In the "golden image" example, you perform all the changes on the actual devices, but that's rarely needed in a Yocto environment where changes are applied during the build on the host machine, and you only provision your target device with the output image.

You can read more about cross-compilers on this Wikipedia post. It's important to build an understanding of this concept for a more seamless experience with Yocto.

The Yocto Project Is a Ground-Up Approach

The starting point in Yocto is to build a distribution image that contains the necessities to boot a system, and that is about it. This is, of course, not very useful on its own, but it's the foundation that you build upon and where only the components that you select to be included are included, and nothing else. This means you're in full control of the distribution that's generated, and it can be tailored for very specific use cases.

Understanding which components make up a Linux distribution might require additional knowledge acquisition. It's not normally something that you piece together when working with binary distributions, as it's already done by someone else. A good reference is the Linux From Scratch project, which is a step-by-step tutorial on how to create your own Linux distribution. It's essentially what Yocto does but in an automated fashion. I don't believe that you need to understand each step that's involved in detail, but you can use the LFS book to get a quick overview of which components can make up a Linux distribution.

The Yocto Project Builds from Source Code

This means instead of working with binary packages, which have been compiled and packaged by someone else, with the Yocto Project, you work with metadata that describes how to build packages from source code. This implies that everything is built from source, including toolchains, Linux kernel images, bootloader images, applications and more.

The workflow in Yocto to install a package onto your custom distribution is

to change the configuration and rebuild. This is the equivalent of running `apt install` on a Debian distribution.

The "build from source" approach is one of the Yocto Project's strengths in the flexibility it provides. Using this approach, you can customize every single package to your needs, and all the changes necessary are applied at build time, which means the output image will contain all the desired customizations—that is a big difference compared to working with a "golden image".

There also are drawbacks to the "build from source" approach. It has significant impact on the time it takes to construct a distribution image, which typically ranges in hours in build time, and involves steps such as fetching source code, unpacking, compiling, installing and so on. Yocto does support a caching mechanism, meaning that subsequent builds will be much faster and are typically in the range of minutes instead of hours.

Because Yocto is a resource-heavy tool, a project using it needs to plan for infrastructure changes and optimizations. This could involve sourcing capable machines to speed up builds or setting up build servers that can be utilized by developers but also where one could run automated builds. There are many optimizations that can be done within Yocto to help speed up builds; you can read more about it in the Yocto Project Mega-Manual.

The Yocto Project—Open-Source License Compliance

It's worth mentioning that open-source license compliance is slightly different in Yocto, because you hold the sources of the produced binaries, in case you need to re-distribute it—for example, for GPL-licensed code. In a binary distribution, the source code of the produced binaries are hosted by the distribution.

For more information, see the section Maintaining Open Source License Compliance During Your Product's Lifecycle in the Yocto Mega-Manual.

Conclusion

Yocto has a steep learning curve, and you should be prepared for it. Yocto has its strength in flexibility, but this also adds to the complexity.

It's fairly easy to do a quick build of a base image, but the hurdle is often moving from this to creating something custom, and to do this, you'll need to spend time reading the Yocto Project Mega-Manual to understand the core concepts. Here are a few topics to start with:

- The layer structuring and overlays, which includes core layers, board support package layers and distribution layers.
- The bitbake build engine and how it parses metadata and executes tasks.
- The Yocto Project workflow.

The investment of learning Yocto does come with benefits that will be valuable in the long-term:

- You will have mastered a tool that gives full insights and control on how you are building the distribution image. The deeper understanding of the system definitely will be helpful when debugging problems once products are in the field.
- You will have a streamlined development workflow with higher automation and a higher level of reproducibility.
- You will have increased re-usability of the software stack if you decide to change hardware or plan to release similar products that are based on the same platform.
- You will gain access to a large community of experienced developers that are

willing to help you along the way.

- You will gain access to a large ecosystem of production-grade software that is easily integrated into your software stack due to Yocto's layering design. ■

Mirza Krak is an embedded Linux solution specialist with the open-source Mender.io project to manage updates for IoT. He has eight years of experience in the field. He is involved in various other open-source projects and is a Linux kernel contributor. Mirza has spoken at various conferences including the Embedded Linux Conferences.

Resources

A large portion of the information and reflections in this article comes from my experience working in the embedded Linux field and my involvement in numerous projects that shipped products based on Linux.

Further resources:

- yoctoproject.org
- Yocto Project Mega-Manual
- Yocto Project Wiki

Decentralized Certificate Authority and Naming

Online Censorship Is Coming—Here's How to Stop It

EU's upload filters are coming. Why and how the Open Source world must fight them.

By Glyn Moody



Glyn Moody has been writing about the internet since 1994, and about free software since 1995. In 1997, he wrote the first mainstream feature about GNU/Linux and free software, which appeared in *Wired*. In 2001, his book *Rebel Code: Linux And The Open Source Revolution* was published. Since then, he has written widely about free software and digital rights. He has a blog, and he is active on social media: @glynmoody on Twitter or identi.ca, and +glynmoody on Google+.

A year ago, I warned about some terrible copyright legislation being drawn up in the EU that would have major adverse effects on the Open Source world. Its most problematic provision would force many for-profit sites operating in the EU to use algorithmic filters to block the upload of unauthorized material by users. As a result of an unprecedented campaign of misinformation, smears and outright lies, supporters managed to convince/trick enough Members of the European Parliament (MEPs) to vote in favour of the the new Copyright Directive, including the deeply flawed upload filters.

A number of changes were made from the original proposals that I discussed last year. Most important, "open source software development and sharing platforms" are explicitly excluded from the scope of the requirement to filter uploads. However, it would be naïve to assume that the Copyright Directive is now acceptable, and that free software will be unaffected.

Open source and the open internet have a symbiotic

relationship—each has fed constantly into the other. The upload filters are a direct attack on the open internet, turning it into a permissioned online space. They will create a censorship system that past experience shows is bound to be abused by companies and governments alike to block legitimate material. It would be a mistake of the highest order for the Open Source community to shrug its shoulders and say: "we're okay—not our problem." The upload filters are most definitely the problem of everyone who cares about the open and healthy internet, and about freedom of speech. For example, the GitHub blog points out that **false positives are likely to be a problem** when upload filters are implemented—regardless of nominal "exemptions" for open source: "When a filter catches a false positive and dependencies disappear, this not only breaks projects—it cuts into software developers' rights as copyright holders too."

So, what can be done?

As the Pirate MEP Julia Reda emphasises in her post summarizing the **multi-year battle to improve the text of the Copyright Directive**: "My message to all who took part in this movement: Be proud of how far we came together! We've proven that organised citizens can make an impact—even if we didn't manage to kill the whole bill in the end. So don't despair!" Specifically:

> A novel alliance of digital rights NGOs, political parties and social media personalities succeeded in politicising and mobilising an entire generation of digital natives. Countless people rose to new challenges: Entertainment YouTubers suddenly found themselves in the role of reporters or political commentators, internet users became activists and organisers, and many participated in the first protests of their lives. These experiences will leave a lasting impact.

That's important, because the concerns and beliefs of that "novel alliance" are closely aligned with those of the Free Software community. The new-found interest in hitherto obscure aspects of the online world and its software are an opportunity for the Open Source world to increase awareness of what it does, and to garner support for its activities. The potential for spreading the word is huge: over **five million people signed an EU petition against upload filters**, and 200,000 took to the

streets to protest. Where new digital rights initiatives are set up to harness the recent mobilization of "digital natives", free software coders can help people understand that open source is a key part of the solution to the problems they seek to address.

Another option is for the community to become involved in the next stage of the fight against upload filters. The legislation just passed by the EU is what is known as a directive; as such, it must be transposed into national law in all the EU's Member States during the next two years. Exactly how that is done, and what the result is, depends on the local legislative process and debate. There is plenty of scope for local variations in the laws implementing the Copyright Directive. This means there will be numerous local battles where open-source organizations can help to blunt the worst aspects of the new law.

Once local transpositions exist, it will be possible to begin the process of challenging upload filters as being inconsistent with existing EU law. There are a number of ways of doing that. When the time comes, the Free Software world can help encourage people to defray the considerable costs of doing so.

Assuming the worst, and that the upload filters survive legal challenges and are widely implemented, there is one final role that open source can—and arguably must—play. The EU Copyright Directive was a deeply dishonest piece of legislation, because it pretended to be about helping artists—a laudable aim. But it was really about attacking Google and, to a lesser extent, Facebook. Its main intent is to force them to pay EU publishers and recording companies, essentially as a punishment for thriving, while the traditional copyright industry is not, largely because of the latter's refusal to embrace fully the digital world and its new possibilities. The EU Copyright Directive is a misguided attempt to turn back the clock and make the internet a tightly controlled, passive medium like television.

In the coming world of the EU Copyright Directive, only big players can afford to participate. The new law will require the licensing of huge quantities of material, and the implementation of complex (and fallible) upload filters to block everything else. Both will be punitively expensive for smaller EU companies. Media power will be

concentrated in the hands of the main copyright players, which will emerge as the gatekeepers and censors in this highly centralized system. A powerful way to fight upload filters and their threat to freedom of speech is to create internet services that are outside this framework—specifically, those that are completely distributed. Eben Moglen, General Counsel of the Free Software Foundation for 13 years, and co-creator of the most recent versions of the GNU GPL, understood this ten years ago. Here's what he said when I interviewed him on the subject in 2010:

> What I am proposing is that we build a social networking stack based around the existing free software we have, which is pretty much the same existing free software the server-side social networking stacks are built on; and we provide ourselves with an appliance which contains a free distribution everybody can make as much of as they want, and cheap hardware of a type which is going to take over the world whether we do it or we don't, because it's so attractive a form factor and function, at the price.

He set up the FreedomBox project to realize that vision. It has been running since 2011, and it has created that "social networking stack" for various single-board computers. Recently, the FreedomBox Foundation launched a plug-and-play commercial version costing 82 euros, widening the reach of the project to non-technical users.

For years, the FreedomBox has been an important if little-known project. Today, in the wake of the disastrous EU Copyright Directive, it is indispensable. Although much good work has already been done by the dedicated FreedomBox coders, key software elements still are missing, and existing programs could be improved. As well as helping to create an alternative to the non-open, filtered internet that is coming to the EU, the FreedomBox initiative will move us closer to creating a more resilient global internet that is censorship-resistant. It should become a priority for the Open Source community. ■

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.